TURING 图灵原创

彭靖田 林健 白小龙◎著



深入理解TensorFlow
架构设计与实现原理


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS



彭靖田，才云科技技术总监，谷歌机器学习开发专家（ML GDE），Kubeflow Core Maintainer，TensorFlow Contributor，曾一度成为TensorFlow社区全球前40的贡献者。加州大学圣迭戈分校访问学者，毕业于浙江大学竺可桢学院求是科学班。曾为华为深度学习团队核心成员，主要参与华为深度学习平台的设计和研发工作。



林健，华为深度学习团队系统工程师。在中科院计算所取得博士学位，并在美国俄亥俄州立大学做过博士后研究。长期从事系统软件研发，工作涉及高性能计算与分布式系统，爱好开源软件与人工智能。曾参与开发CNGrid GOS、MVAPICH等工业级软件，并合作创建LingCloud、DataMPI等开源项目。



白小龙，华为公司深度学习云服务的技术负责人，主要负责深度学习平台、模型和算法的研发。长期从事信号、图像处理和机器学习研究，于2015年6月毕业于浙江大学并取得工学博士学位，曾获教育部博士生学术新人奖。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

TURING 图灵原创

彭靖田 林健 白小龙◎著



深入理解 TensorFlow

架构设计与实现原理

人民邮电出版社
北京

图书在版编目（C I P）数据

深入理解TensorFlow ：架构设计与实现原理 / 彭靖田，林健，白小龙著. -- 北京 ：人民邮电出版社，2018.5（2018.10重印）
（图灵原创）
ISBN 978-7-115-48094-1

Ⅰ. ①深… Ⅱ. ①彭… ②林… ③白… Ⅲ. ①人工智能—算法—研究 Ⅳ. ①TP18

中国版本图书馆CIP数据核字(2018)第049742号

内 容 提 要

本书以 TensorFlow 1.2 为基础，从基本概念、内部实现和最佳实践等方面深入剖析了 TensorFlow。书中首先介绍了 TensorFlow 设计目标、基本架构、环境准备和基础概念，接着重点介绍了以数据流图为核心的机器学习编程框架的设计原则与核心实现，紧接着还将 TensorFlow 与深度学习相结合，从理论基础和程序实现这两个方面系统介绍了 CNN、GAN 和 RNN 等经典模型，然后深入剖析了 TensorFlow 运行时核心、通信原理和数据流图计算的原理与实现，最后全面介绍了 TensorFlow 生态系统的发展。

本书适合想深入了解 TensorFlow 的程序员。

◆ 著　　彭靖田　林　健　白小龙
　责任编辑　王军花
　责任印制　周昇亮
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　印刷
◆ 开本：800×1000　1/16
　印张：23　　　　彩插：2
　字数：544千字　　　　2018年 5 月第 1 版
　印数：6 001 – 7 000册　　　　2018年 10 月北京第 4 次印刷

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

序　一

人工智能迎来了继 20 世纪 90 年代以来的又一次大发展，人工智能和深度学习无疑是近年来最受追捧的热点。2016 年和 2017 年 AlphaGo 两次分别战胜韩国和中国围棋国手，更是让人工智能成为街头巷尾的热谈。为什么人工智能在经历了两起两落后再一次迎来了复兴？我认为，这一次的爆发主要由理论、应用、硬件和软件四个方面的原因促成的，貌似偶然实则必然。

Geoffrey Hinton 等人在 2006 年发现了训练高层神经网络的有效算法，为深度学习的研究打开了新局面，这也是人工智能得以重燃的导火线。经过后续研究人员的努力，尤其是 CNN 和 RNN 的出现，深度学习、神经网络方法在图像和语音识别方面显示出非常好的效果，大大超越之前的理论和方法，甚至能够突破人类极限。与计算机视觉、机器人、自然语言理解、信息检索等技术相结合，深度学习的应用也从单纯的图像和语音识别扩展到自动驾驶、图像增强与风格替换、文本语音间转换和推荐系统等。使用深度学习技术的应用和初创企业如雨后春笋般冒了出来。尤其需要指出的是，互联网和大数据的广泛应用是深度学习发展的必要条件。一般而言，数据量越大且数据质量越高，由深度学习训练出来的模型精度也就越好。计算机硬件的发展也直接推动了深度学习的发展。GPU 和其他专用加速器件的出现，大大提高了深度学习的计算效率。在语音识别场景下，GPU 可将数十亿样本的训练时间从数年缩短到数天。而专用的 ASIC 加速芯片比 GPU 的能效更有一个数量级以上的提升，也让深度学习从服务器端走向手机端，进一步拓展了其应用范围。

开放的软件生态和易用的软件形态是形成人工智能和深度学习产业链至关重要的两个方面。没有软件的支撑，理论很难与应用相结合，新硬件也很难为应用提速。从大数据软件的发展历程可以想见，如果没有开源的 Hadoop 生态系统，以及受其设计思想影响的新型大规模并行处理数据库系统，我们现在可能还在为如何管理和处理 PB 乃至 EB 级的数据发愁。开源 TensorFlow 的出现解决了类似的问题，一下子拉近了深度学习理论与实际应用的距离。同时，TensorFlow 也具备迈向成功生态系统的必要条件，即差异化的软件功能、刚需的典型应用和活跃的社区支持，发展前景可期。但是，原生 TensorFlow 的软件形态尚不足以支持深度学习的全流程生产化应用，欠缺诸如数据管理和预处理，模型训练、管理和运行，资源管理、任务调度和运行时监控等能力，导致最终用户形成生产力的成本过高。深度学习是计算密集型重资产类应用，如果有能够提供异构高性能计算资源并能够集成上述平台化功能的深度学习公有云服务，可降低 TensorFlow 的使用门槛并提升用户体验，客观上会与开源效应叠加，起到加速产业发展的作用。

很高兴能够在这个时候看到一本讲述如何使用 TensorFlow 的专业图书。作者是深谙计算机系统之道的一线工程师，带给读者的是基于实战经验的理解。非常难得的是，本书除了讲解如何使用 TensorFlow，还加入了对系统设计原理方面的剖析，有助于读者做针对性的应用和系统优化。相信本书对从事深度学习方面研究和开发的读者定会有所裨益。

查礼
中国科学院计算技术研究所副研究员
中国大数据技术大会（BDTC）发起人

序　二

如果说要评选出一个在过去一年中学术界和产业界最热门的词汇，无疑非“人工智能”莫属。作为新一轮产业变革的核心驱动力，人工智能正重构生产、分配、交换、消费等经济活动各环节，形成从宏观到微观各领域的智能化新需求、新产品、新技术、新业态，引发经济结构重大演化，实现社会生产力的整体跃升。

人工智能系统的智能有三种来源。一种是依靠人类设计者的知识输入，为系统建立人工特征、知识库和推理机制，传统意义上的专家系统属于这一范畴。二是通过数据驱动的归纳式学习，近年来大火的深度学习即属于这一类。与依赖于人工经验、通过手工构建的知识特征不同，深度学习以端到端的方式进行特征学习，其基本动机在于构建多层网络来学习隐含在数据内部的关系，从而使学习得到的特征具有更强、更泛化的表达能力。三是智能体通过与环境交互，学习经验和知识并更新知识表示。近年来，以深度学习为代表的数据驱动方法在图像识别、语音识别、机器翻译、自然语言理解等任务中取得了一系列突破。人脸识别、自然语言理解在一系列国际评测中展示了超越人类能力的水平，语言识别和机器翻译也达到了一个前所未有的高度。

在我看来，各行各业还会在相当长的一段时间内享受到基于大数据的深度学习红利。将深度学习红利释放到读者所在的行业是提升行业智能水平的一条捷径。而要做到这一点，核心关键在于要降低大数据深度学习技术的使用门槛。要使看起来高深的深度学习技术，早日达到“旧时王谢堂前燕，飞入寻常百姓家”的程度，开源软件生态社区会起到巨大的推动作用，TensorFlow 则无疑正是在这方面的佼佼者。深度学习对于张量计算性能、算子灵活性、自动微分能力、分布式训练、自动调参数、可视化和端侧部署等都有很强的诉求，而 TensorFlow 的设计也充分考虑到了这些因素，这使得它成为了当前业界很流行的一个深度学习引擎。

TensorFlow 还是一个较新的技术，但是发展极为迅猛，这时候出现一本深入浅出讲解 TensorFlow 理论与应用的图书，对于希望学习和应用大数据深度学习技术的广大读者而言，诚“如大旱之望云霓”。本书理论与实践并重，理论上讲清楚了一些本质的东西，并加入了作者对系统设计原理方面的深刻理解，进而通过实际案例，引导读者掌握针对性的系统优化的技能。

本书第一作者是我的学生，2012 年入学时进入了浙大计算机专业的尖子班“求是科学班”，我担任了他们这个班的班主任。他不仅品学兼优，而且作为班上的团支书，帮我这个不太称职的班主任做了很多班级工作。在我心目中，他依然是入学时的青涩模样，转眼间却已成为开源软件

界的技术翘楚，作为老师，最欣慰的莫过于此了吧。是为序。

陈刚
教育部“长江学者”特聘教授
浙江大学计算机学院院长

前　　言

缘起

2016 年的某个中午，我在知乎上回答了名为“如何高效学习 TensorFlow 代码？”的问题，其中简单介绍了我在 TensorFlow 开源社区的贡献，以及 TensorFlow 的学习路线和方法。此回答引起了一些圈内人的共鸣。人民邮电出版社图灵公司的编辑也在第一时间找到我，希望我能够写一本 TensorFlow 相关的图书。于是，便有了你手中的这本书。

为什么写本书

在 ImageNet 的带动下，深度学习的研究热潮已席卷全球。随着 AlphaGo 的横空出世，资本市场对人工智能的产业化也表现出了空前的兴趣。2017 年 7 月，《国务院关于印发新一代人工智能发展规划的通知》的出台，标志着国家层面对人工智能发展的高度重视，明确了我国新一代人工智能发展的战略目标。未来 10 年，我们将见证人工智能全面升级改造传统行业。在这场深刻变革中，TensorFlow 将迸发出巨大的能量。

TensorFlow 推出短短一个月，就成为了机器学习和深度学习项目中最受欢迎的开源框架。究其原因，离不开 Google 在人工智能与数据处理领域的深厚积淀及其在业界的强大号召力。TensorFlow 自 2015 年 11 月开源以来，已经发布了 30 多个版本。尽管 TensorFlow 整个生态系统是开源的，但由于它版本升级过快，且算子种类众多，大部分公司，尤其是一些中小型或创业公司，难以在有限的时间内快速掌握 TensorFlow 的设计思想和基本原理。TensorFlow 开源的 PS-worker 分布式架构也在快速迭代，与其内部基于 Borg 调度的分布式架构并不相同。自 TensorFlow 开源以来，不断有人撰写图书或博客解释其各组件的实现与原理。但遗憾的是，能够深入剖析 TensorFlow 内部实现细节与设计思想的资料少之又少，而本书则尝试弥补这一缺憾。

本书以 TensorFlow 1.2 为基础，从基本概念、内部实现和最佳实践等方面深入剖析了 TensorFlow。书中重点阐述了以数据流图为核心的机器学习编程框架的设计原则与核心实现，并且介绍了 TensorFlow 生态系统中的两大重要工具：TensorBoard 可视化工具与 TensorFlow Serving 模型托管工具。同时，本书还将 TensorFlow 与深度学习相结合，从理论基础和程序实现这两个方面系统介绍了卷积神经网络（CNN）、生成对抗网络（GAN）和循环神经网络（RNN）等经典模型。本书不仅由浅入深地全面介绍了 TensorFlow 的使用方法，而且结合源代码进行了深入

剖析，使读者可以快速、系统地学习 TensorFlow 的架构设计与实现原理。

读者对象

本书的读者主要包含以下人员。

- **TensorFlow 二次开发人员**。由于在高效性、多平台、多语言、稳定性等方面的诸多优点，TensorFlow 已被国内外越来越多的公司采用并部署到生产环境。而为了解决特定场景下的特定问题，大部分公司选择在开源 TensorFlow 的基础上进行二次开发。通过这本书，这部分人员可以深入而又全面地了解 TensorFlow 的设计原则和实现细节，这是修改 TensorFlow 内核的前提。
- **数据科学家和算法工程师**。如果要使用 TensorFlow 解决生产和生活中的实际问题，仅掌握 TensorFlow 基本使用方法是远远不够的，还必须对 TensorFlow 的设计理念、架构和运作机制有一定了解。尤其是对于分布式训练任务，更需要深入了解 TensorFlow 分布式的架构设计与多种并行模式的实现原理。对于这部分读者来说，本书将带领他们走入 TensorFlow 架构师的内心世界，使其系统、深入地理解 TensorFlow 和数据流图，提高开发水平，从而编写出更加高效的深度学习和机器学习模型。
- **人工智能方向的研究生**。对于一名人工智能专业的研究生来说，除了需要具备扎实的人工智能理论功底外，还应当熟练掌握一种算法模型编程框架，才能将研究课题中的问题快速落实到实际的代码上来。而 TensorFlow 便是当下最受欢迎的机器学习和深度学习框架。通过阅读本书，人工智能方向的研究生可以全面提升复现论文实验结果和开发全新模型的效率，并深入理解 TensorFlow 的设计思想和实现细节。
- **开源软件爱好者**。TensorFlow 是全世界最受欢迎的开源机器学习和深度学习框架，它在设计和实现过程中参考了 Google 第一代分布式机器学习框架 DistBelief 的实践经验，同时又加入了很多值得学习的创新。本书分析 TensorFlow 架构设计和实现原理的方式也许值得许多开源软件爱好者学习和借鉴，这部分读者不仅能够领略到开源软件的优秀设计，还可以掌握分析开源软件源代码的方法和技巧，从而进一步提高使用开源软件的效率和质量。

如何阅读本书

本书分为五大部分（不包括附录）。

第一部分为基础篇（第 1 ~ 3 章），简单介绍了 TensorFlow 设计目标、基本架构、环境准备和基础概念，包括数据流图的设计与使用，以及 TensorFlow 运行环境和训练机制，帮助读者快速入门 TensorFlow，迅速上手使用。

第二部分为关键模块篇（第 4 ~ 7 章），着重讲解了使用 TensorFlow 端到端解决人工智能问题涉及的关键模块，包括数据处理、编程框架、可视化工具和模型托管工具，帮助读者进一步提

升开发效率，快速落地模型应用。

第三部分为算法模型篇（第 8 ~ 11 章），在读者熟练掌握 TensorFlow 后，该部分将深度学习与 TensorFlow 有机结合，系统介绍了深度学习的发展历史与应用场景，并结合理论与代码实现深入讲解了 CNN、GAN 和 RNN 等经典模型。

第四部分为核心揭秘篇（第 12 ~ 14 章），深入剖析了 TensorFlow 运行时核心、通信原理和数据流图计算的原理与实现，聚焦 C++ 核心层的揭秘，帮助读者进一步理解 TensorFlow 底层的设计思想与实现细节，TensorFlow 二次开发人员需重点关注这部分内容。

第五部分为生态发展篇（第 15 章），全面介绍了 TensorFlow 生态系统发展，并重点介绍了 Keras 深度学习算法库，以及 TensorFlow 与云原生社区 Kubernetes 生态的结合、与大数据社区 Spark 生态的结合，并介绍了 TensorFlow 通信优化技术、TPU 及 NNVM 模块化深度学习技术，帮助读者进一步全面了解深度学习生态发展的现状。

勘误和支持

感谢您在茫茫书海中选择了我们的作品。尽管我们在写作过程中力求以精确的语言传达正确的信息，然而由于专业水平与表达能力的限制，书中难免仍有不周之处。深度学习相关理论与技术演进飞速，书中部分内容在呈现于读者之时或许已不合时宜。我们诚挚地希望你能够指正本书存在的问题，并给我们提出宝贵的意见和建议。

读者可以通过图灵社区的本书主页（http://www.ituring.com.cn/book/2397）提交或查看勘误。本书的样章与部分示例代码也已托管在 GitHub 项目（https://github.com/DjangoPeng/tensorflow-in-depth）中，欢迎读者访问并提出意见与建议。

致谢

感谢我的合作者林健博士，他在计算机系统方面学识渊博，使我在合作著书的过程中不断进步。同时，林健博士对本书的审稿和校稿工作也做出了重要贡献。

感谢我的另一位合作者白小龙博士，他对深度学习理论和落地的丰富经验使我受益匪浅。

感谢人民邮电出版社图灵公司的王军花编辑在这一年多的时间中始终支持我们的写作，她的鼓励和帮助使我们顺利完成了本书。

感谢丁泽震、辛现银、张震宇、李鹏飞、尉建等朋友在写书和校稿过程中给我们提供的修改建议与各种帮助。

谨以此书献给我们最亲爱的家人，以及众多热爱 TensorFlow 和深度学习的朋友们！

彭靖田

于杭州

作者简介

彭靖田

才云科技技术总监，才云人工智能云平台 Clever 负责人。谷歌机器学习开发专家（ML Google Developer Experts）。Kubeflow Core Maintainer，作为创始团队与 Google 一起打造云原生机器学习社区——Kubeflow。TensorFlow Contributor，曾一度成为 TensorFlow 社区全球前 40 的贡献者。加州大学圣迭戈分校访问学者，毕业于浙江大学竺可桢学院求是科学班。曾为华为公司深度学习团队核心成员，主要参与华为深度学习平台的设计和研发工作。从高中开始研究算法，并通过 OI 获奖保送至浙江大学，兼具算法工程师、系统工程师和云计算工程师的多重视角，拥有整合人工智能平台、框架和算法模型的丰富经验。

林健

华为公司高级软件工程师。在中国科学院计算技术研究所取得博士学位，并在美国俄亥俄州立大学做过博士后研究。长期从事系统层软件研发，工作方向涉及高性能计算、分布式系统与大数据处理，爱好开源软件与人工智能技术。曾参与开发国家科学计算基础设施——中国国家网格 CNGrid 的核心系统软件 GOS，以及工业界广泛使用的高性能 MPI 通信库——MVAPICH2。以核心开发者身份合作创建开源云服务系统 LingCloud 和开源数据计算库 DataMPI。设计实现教学型中国象棋智能算法引擎 BitStronger，并多次参加计算机棋类博弈赛事。参与翻译《C++ API 设计》《C++探秘：68 讲贯通 C++》等技术图书。

白小龙

华为公司深度学习云服务的技术负责人，主要负责深度学习平台、模型和算法的研发。于 2015 年 6 月毕业于浙江大学并取得工学博士学位，曾获教育部博士生学术新人奖。长期致力于研究面向工业应用的信号处理、图像处理和机器学习算法。曾提出基于图像显著性的大规模缺陷目标快速检测算法用于工业智能检测，并与新加坡国立大学合作研发基于医疗图像的青光眼特征选择和多标签学习算法及软件。相关论文发表于 *IEEE Transactions on Industrial Informatics*、*Computer Methods and Programs in Biomedicine* 等 SCI 期刊。

目　录

第一部分　基础篇

第二部分　关键模块篇

第三部分　算法模型篇

第四部分　核心揭秘篇

第一部分

基础篇

- ❑ 第 1 章　TensorFlow 系统概述
- ❑ 第 2 章　TensorFlow 环境准备
- ❑ 第 3 章　TensorFlow 基础概念

第 1 章

TensorFlow 系统概述



人工智能和深度学习的热潮将 TensorFlow 推向了很高的地位，媒体的追捧和业界的宣传也为这一源自 Google 的开源软件增添了传奇的色彩。对于技术从业者或爱好者而言，我们初识 TensorFlow 时有必要拨开表象看本质。本章作为引子，首先从技术视角概括性地介绍 TensorFlow 的产生背景、独特价值、版本变迁，以及它与其他主流深度学习框架的异同。同时，本章从灵活通用性、异构支持性和性能高效性三个视角解析 TensorFlow 的设计目标，展示 TensorFlow 作为一款兼具深度学习库、人工智能引擎和基础平台软件身份的开源产品的优势所在。最后，我们将简单介绍 TensorFlow 的工作形态和组件结构，帮助读者快速建立对 TensorFlow 软件架构的第一印象。

1.1 简介

当今，人工智能领域最受欢迎的深度学习和机器学习框架非 Google 开源的 TensorFlow 莫属。本节我们将依次介绍 TensorFlow 的产生背景、独特价值和版本变迁，并横向对比目前主流机器学习和深度学习框架各自的特点和优劣。

1.1.1 产生背景

随着近年来深度学习模型在图像、视觉和语音领域不断取得突破，相关研究热潮持续高涨，开源深度学习框架出现百花齐放之势，其中具有代表性的框架包括 XGBoost、Theano、Torch、Caffe 和 MXNet 等。它们有的计算速度快，有的可移植性好，有的内存占用少，有的易于上手。在 Google 推出 TensorFlow 之前，大家仍处于“你方唱罢我登场”的百家争鸣状态。然而，Google 推出深度学习框架 TensorFlow 之后，江湖巨变。

2015 年 10 月，Google 旗下的 DeepMind 公司研发的 AlphaGo 击败樊麾，成为第一个无须让子即可在 19 路棋盘上击败职业围棋选手的电脑程序。这一壮举不仅打破了人工智能无法在围棋领域战胜人类顶尖棋手的诅咒，而且吸引了全球各界对人工智能研究的高度关注。2016 年 1 月，AlphaGo 的研究成果发表在知名学术期刊《自然》上，这一事件也将人工智能热潮推向了新的高度。2017 年 5 月，强化后的 AlphaGo 与世界围棋三甲之一的柯洁对弈，获得 3：0 全胜战绩。这

再一次笃定了人们对于 Google 在人工智能领域遥遥领先的信念。

AlphaGo 的后期版本使用了基于 TensorFlow 编写的算法模型。但事实上，AlphaGo 只是让 TensorFlow 走进公众视野的一个契机，TensorFlow 的原始动机则是 Google 在高速发展的信息化应用背景下，发展感知、预测等人工智能技术的需求。移动互联网、物联网、共享经济、增强实现……这些热词的背后无一不需要海量数据与智能处理能力的支撑。作为行业的引领者，Google 公司内部很早便有自研的机器学习平台。TensorFlow 是既有平台的多年技术积累在新的时代背景下蜕变升华的成果。

TensorFlow 推出后短短一个月，就成为了机器学习和深度学习项目中最受欢迎的开源框架。究其原因，离不开 Google 在人工智能与数据处理领域的深厚积淀及其在业界的强大号召力。同时，Google 已经成功领导了多个开源项目，典型的有移动操作系统 Android、容器编排引擎 Kubernetes、编程语言 Go 等，它们充分体现了 Google 的工程水准与协作精神。因此，在内因与外因的合力之下，TensorFlow 的横空出世也就不难理解了。

1.1.2 独特价值

TensorFlow 能够在众多开源框架中杀出重围，除了 Google 的背书以外，一定有其自身的独特价值。下面重点介绍 TensorFlow 相比其他开源框架的亮点和优势。

- **运算性能强劲**。在构建和部署机器学习系统时，性能是至关重要的。TensorFlow 1.0 加入的线性代数编译器 XLA 全方位地提升了计算性能。XLA 可以帮助 TensorFlow 在 CPU、GPU、TPU、嵌入式设备等平台上更快速地运行机器学习模型的训练与推理任务。同时，TensorFlow 提供了大量针对不同软硬件环境的优化配置参数。用户可以根据自身的需求和应用的特点，进一步提升计算性能。
- **框架设计通用**。TensorFlow 并非只是纯粹的神经网络库。TensorFlow 最初由 Google Brain 小组（隶属于 Google 机器智能研究机构）的研究员和工程师们开发出来，用于机器学习和深度神经网络方面的研究，但其灵活的设计也可广泛用于其他计算领域。同时，TensorFlow 既提供高层封装 API（如 Slim、Keras、TF Layers 等），能够帮助用户快速实现算法原型；又提供底层原生 API，可以实现更灵活且高效的分布式并行模式。
- **支持生产环境部署**。TensorFlow 支持使用同一套 API 实现探索环境和生产环境的部署。曾经，科研人员将算法原型推广到生产环境中使用的过程非常痛苦，因为这涉及大量的模型重写和脚本适配工作。现在，使用 TensorFlow 的算法研发人员既可以快速地将想法和原型运用到生产环境的产品中，也可以在学术圈更方便地分享自己的研究成果。
- **语言接口丰富**。TensorFlow 核心层由 C++ 实现，应用层使用 SWIG 等技术封装，提供了多语言 API 的支持。目前，官方支持的语言有 Python、C、C++、Java、Go 等。除此之外，TensorFlow 的社区贡献者们也提供了非官方的应用层 API，如 Node.js（https://github.com/node-tensorflow/node-tensorflow）、Julia（https://github.com/malmaud/TensorFlow.jl）、R（https://github.com/rstudio/tensorflow）。

- **端云协同计算**。TensorFlow 同时支持在云侧（服务器端）和端侧（移动设备等终端）运行，有效结合了云侧和端侧的各自优势。在云侧方面，TensorFlow 提供多种并行模式和编译优化等技术，尽可能提升算法模型的运算性能；在端侧方面，TensorFlow 提供轻量级部署和 8 比特压缩等技术，尽可能提升计算和存储资源利用效率。

除以上列举的优势外，TensorFlow 丰富的算子库和教学资料也是其独有的竞争优势。同时，TensorFlow 社区的活跃度遥遥领先其他竞争者，每个月都会有上万行的代码合入主分支，这使得 TensorFlow 的新特性能够快速实现，bug 也能快速修复。我们相信在业界众多人工智能开发者和 Google 工程师的共同努力下，TensorFlow 能够计算得越来越快、发展得越来越好。

1.1.3　版本变迁

TensorFlow 自 2015 年 11 月开源以来，已经发布了 30 多个版本。本节从 TensorFlow 的发展历程入手，考查其关键特性的发布和对应版本的变迁。图 1-1 展示了这一变化过程。

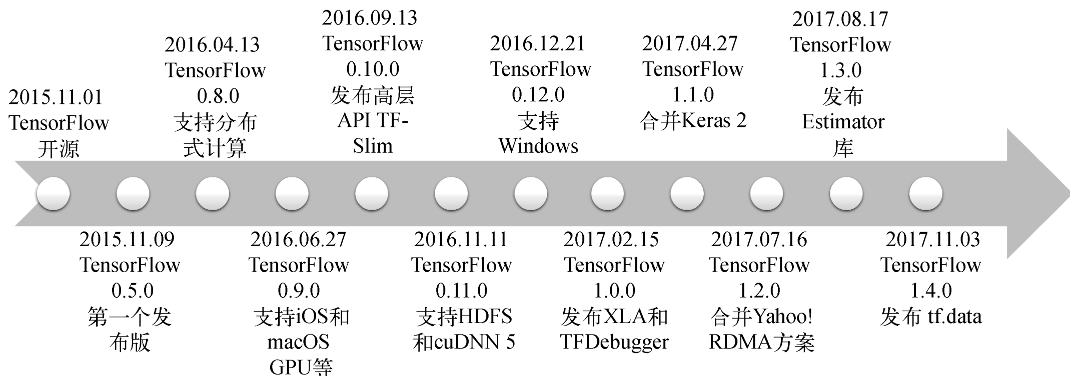


图 1-1　TensorFlow 关键特性发布和对应版本变迁

在 TensorFlow 开源后第 9 天，Google 带来了第一个正式发布版——TensorFlow 0.5.0，不过该版本仅支持在 Linux 系统上运行单机模型。随着 TensorFlow 贡献者和用户对分布式的呼声越来越高，2016 年 4 月发布的 TensorFlow 0.8.0 开始初步支持分布式计算。两个月后，TensorFlow 0.9.0 增加了对多平台的支持。自此，用户可以将 TensorFlow 部署在 iOS 和树莓派（Raspberry Pi）上。同时，该版本还支持在 macOS 上使用 GPU 运行算法模型。

2016 年 9 月，0.10.0 版本的发布解决了 TensorFlow 学习成本高和上手难的问题。尤其对于非计算机背景的算法研究人员，使用该版本提供的高层 API——TF-Slim 能够快速实现图像和视觉领域的算法模型。TF-Slim 的发布有效扩大了 TensorFlow 用户群体，使得高校和科研院所的研究者们也能够享受 TensorFlow 带来的便利。

随着 TensorFlow 开源一周年而到来的 0.11.0 版本，新增了对 HDFS 和 cuDNN 5 的支持。HDFS 作为 Hadoop 生态中的分布式文件系统，广泛地应用于大数据系统，这一特性标志着大数

据生态和 TensorFlow 开始相互合作和共赢发展；cuDNN 是 NVIDIA 公司开发的深度神经网络库，cuDNN 5 能够进一步提升神经网络任务在 GPU 硬件上的计算速度。紧接着发布的 TensorFlow 0.12.0 开始为 Windows 平台提供支持，同时提供了实验性的 Go 语言应用层 API。

2017 年 2 月，Google 在山景城（Mountain View）召开了 TensorFlow Dev Summit 2017 大会。大会全面介绍了 TensorFlow 的进展和取得的成就，并于隔天发布了 TensorFlow 1.0.0。这也成为了 TensorFlow 发展的一个重大里程碑事件，标志着 TensorFlow 已经初步成熟并能够支持生产环境部署。事实上，当时京东、小米、Uber 等国内外公司也确实在生产环境中使用了 TensorFlow。作为 TensorFlow 的第一个正式版，TensorFlow 1.0.0 带来了诸多提升性能和易用性的关键特性，比如：线性代数编译器 XLA，部分解决了内存消耗大和计算速度慢的问题；命令行调测工具 TensorFlow Debugger，初步解决了算法模型调测困难的问题。同时，该版本还新增了对 Android 的友好支持，使得用户能够更快速地将 TensorFlow 编写的模型部署到移动设备上运行。

2017 年 4 月，TensorFlow 1.1.0 将 Keras 2 合并到了项目主分支的 tf.contrib.keras 目录中。从此以后，用户再也不需要独立安装 Keras 软件包，TensorFlow 将自带 Keras API。Keras 是一套类似于 TF-Slim 的高层 API，它良好的封装性和对模型的高度抽象使之收获了一大批算法开发者。但是，Keras 并不等于 TensorFlow，诸如分布式运行和更灵活的计算模式等还得使用 TensorFlow 原生 API 实现。同时，该版本还支持用户在 Windows 上使用 Java 语言的应用层 API。

2017 年 7 月，TensorFlow 1.2.0 正式合入了 Yahoo! 提供的面向 InfiniBand 等高性能网络的 RDMA 通信方案。早在 TensorFlow 白皮书中，Google 就表示 TensorFlow 支持 RDMA。但可能出于商业考虑或其他原因，一直没有将 RDMA 方案发布到 TensorFlow 开源版本中。直到 TensorFlow 1.2.0 发布，用户终于可以在高性能网络设备上享受 RDMA 带来的效率提升，这有效解决了分布式训练大模型时的通信瓶颈问题。经测试，在 VGG 等大模型的分布式训练场景下，相比 TCP/IP，RDMA 能够减少一半左右的网络通信开销。

2017 年 8 月，TensorFlow 1.3.0 发布。它允许用户使用新增的 Estimator 库，以开箱即用方式快速实现深度神经网络分类器（DNNClassifier）、深度神经网络回归器（DNNRegressor）、线性分类器（LinearClassifier），以及深度神经网络和线性混合分类器（DNNLinearCombinedClassifier）。同时，从 TensorFlow 1.3.0 开始的所有二进制发布包都默认使用 cuDNN 6，这将进一步提升 TensorFlow 在 GPU 上的运算性能。

2017 年 11 月，TensorFlow 1.4.0 发布。该版本新增了 tf.data 模块，为数据读入和处理提供了便捷高效的解决方案。该版本还增强了 Estimator 的能力，使其能够支持简单的分布式模型训练和评估。同时，Google 还开源了 GANEstimator 库，以回应越来越多的用户对生成对抗网络（GAN）的需求。

回望 TensorFlow 的发展历程，我们不难发现 Google 在 TensorFlow 项目的推进上投入了不小的资源。在 TensorFlow 项目组成员和贡献者的共同努力下，TensorFlow 正一步步走向成熟。

1.1.4　与其他主流深度学习框架的对比

放眼全球，诸如 Google、Facebook、Amazon 和 Microsoft 等国际巨头均在深度学习领域着手布局。一时间，江湖风云四起，各大门派争相斗法。Google 坐拥 TensorFlow，捍卫江湖地位；Facebook 携手 Caffe2 和 PyTorch，以图三分天下；Amazon 拥抱 MXNet，不甘落于人后；Microsoft 坚守 Cognitive Toolkit（CNTK），寻求单点突破。除此之外，还有 Caffe、Torch7、Theano 等老一辈深度学习框架参与竞争。可谓是乱花渐欲迷人眼，用户不知如何选。本节将为用户客观分析各大主流深度学习框架的特点与优劣。

下面我们从社区活跃度、多语言支持、教学资源、运算性能等多个维度全方位对比主流的深度学习框架。考虑到 Keras 的广大用户基础，我们也将其单独列出来进行比较。表 1-1 对比了 2017 年 12 月各个深度学习框架在 GitHub 上的统计数据。不难发现，TensorFlow 在各项指标中均遥遥领先。排名第二的 Keras 由于接口简单易用而受到广泛关注，目前 Keras 官方已经支持使用 TensorFlow、Theano 和 CNTK 作为后端计算引擎。紧随其后的是老牌深度学习框架 Caffe，其创始人是加州大学伯克利分校的贾扬清博士，他同时也参与了 TensorFlow 项目的早期设计和实现。

表 1-1　主流深度学习框架在 GitHub 的上统计数据

框架名称	所属机构	多语言支持	Star 数量	Fork 数量	贡献者数量
TensorFlow	Google	Python/C/C++/Java/Go	84132	41072	1226
Keras	keras-team	Python	23530	8582	590
Caffe	BVLC	Python/C++/Matlab/Cmd-line	22024	13509	253
CNTK	Microsoft	Python/C++/C#	13488	3522	158
MXNet	Amazon	Python/C++/R/Scala/Julia	12594	4641	465
PyTorch	Facebook	Python	10737	2225	375
Torch7	Facebook	Lua	7575	2222	133
Theano	蒙特利尔大学	Python	7515	2370	326
Caffe2	Facebook	Python/C++	6665	1510	140

从目前局势来看，TensorFlow 是最受欢迎的深度学习框架。那么，抛开 Google 在行业的巨大影响力，TensorFlow 自身的硬实力如何呢？我们参考了多种公开基准测评，以及我们在图像和视觉领域实际测试得到的数据，给出了表 1-2 这组相对客观的横向对比。表中各评价指标均为 5 分制。因为部分框架未能找到测试数据，所以表中仅列出了指标相对确定的深度学习框架。

表 1-2　主流深度学习框架在各个维度的横向对比

框架名称	教学资源	多语言接口	模型设计	运算性能	易用性
TensorFlow	5	5	5	4	5
MXNet	3	5	4	5	4
Caffe	2	5	3	5	3
Theano	3	2	5	3	2
Torch7	2	2	5	3	3
CNTK	2	3	2	5	3

对于深度学习框架的初学者来说，教学资源是一个非常重要的参考指标。借助 Google 的强大影响力和执行力，TensorFlow 在这方面具有显著优势。不论是基础的指导手册，抑或是花样百出的最佳实践，初学者都有大量资料可以查询。同时，人工智能相关的会议和期刊论文中发布的新模型和新算法，几乎都会有人第一时间使用 TensorFlow 实现，并在 GitHub 上开源出来。

从多语言接口的角度来看，TensorFlow 和 MXNet 共同占据领先地位。几乎所有框架都支持深度学习领域的“英语”——Python。TensorFlow 对更多不同编程范式语言的支持使之对于不同背景的用户都具有一定的友好性，同时也扩展了框架潜在的应用领域。

从模型设计维度来看，TensorFlow 采用了当前主流的基于数据流图的模型设计方式。其算子种类丰富，粒度较细，为用户提供的自由度高。相比于 Caffe 的配置式模型设计，TensorFlow 显得更加灵活，能够适应更多的应用场景。同样使用数据流图定义模型的还有 MXNet。不过，MXNet 的分布式模型的约束较多，灵活性不足。

自 TensorFlow 发布以来，运算性能似乎一直是其弱项。在 GitHub 和 Stack Overflow 上的讨论帖中，我们也时常能够看到有人对 TensorFlow 的内存消耗和计算速度表示遗憾。但是，随着 XLA 和 RDMA 等特性的发布，TensorFlow 的性能在绝大多数情况下都不输于其他深度学习框架。如果用户能够深入了解 TensorFlow 的 API，那么就会发现它提供了大量提升性能的配置项。在启用这些性能优化选项后，TensorFlow 的运算性能甚至能够超过 MXNet 和 Caffe。

TensorFlow 的灵活性是它的一大优势，但同时也因为 API 过于丰富而带来了学习成本高的问题。尤其是对于仅研究算法和模型的开发人员，在没有时间全面了解 TensorFlow 运行机制和编程接口的前提下，往往觉得无从下手。针对这类用户，社区的开发者们也提出了不少解决方案，那就是以 Keras、TF Layers 和 TF Learn 等为代表的高层 API 封装库。这些库隐藏了 TensorFlow 的大量细粒度接口，以简单易懂的接口取而代之，使得读者能够快速上手编程，并实现一些单机运行的算法模型。

综合对比当前主流的深度学习框架，TensorFlow 在各个维度都具有比较明显的优势。同时，TensorFlow 社区的活跃度也远超其他社区，这会使得越来越多的深度学习从业者参与贡献 TensorFlow 项目，最终形成越用越好用的良性循环。

1.2　设计目标

作为 Google 公司上一代深度学习平台 DistBelief 的继任者，TensorFlow 的首要设计目标是满足公司内部的图像、语音、语义等感知和预测类应用的需求。这些应用的数据规模和模型复杂度日益增长，对提供计算能力的软硬件平台的功能和性能不断提出更高的要求。与此同时，新一波人工智能浪潮的兴起也促使在开源软件商业生态建设方面经验丰厚的 Google 公司敏锐嗅探信息化、智能化产业发展的新动向，并广泛吸收工业界、学术界和开源社区的前沿理念与最佳实践，将这些思想纳入 TensorFlow 的设计。种种迹象表明，TensorFlow 的设计目标并非局限一套深度学习库，Google 希望其成为一套面向多种应用场景和编程范式、支持异构计算平台、具备优异性

能与可伸缩性的通用人工智能引擎。本节从中选取几个侧面，对 TensorFlow 的设计目标进行介绍和分析。

1.2.1　灵活通用的深度学习库

近几年来，随着海量数据的涌现、硬件计算能力的提升以及神经网络模型和机器学习算法的改进，深度学习技术得到了快速发展，已经成为学术界和工业界共同关注的热点。一方面，深度学习模型和算法的理论研究还未完全成熟，其发展空间巨大，吸引了大量科研人员参与其中。这几年深度学习席卷了各大顶级学术会议，模型设计、方法优化和应用创新的迭代更新速度极快。另一方面，深度学习在某些领域已经可以落地，比如人脸识别模型成功应用于安防系统，语音识别模型成功用于智能终端。除此之外，工业界正在积极探索深度学习更多潜在的商业应用场景。在人工智能的热潮中，很多开源的深度学习库应运而生，这对加速相关研究和工程化效率起到了非常重要的作用。为了应对上述研究与应用领域的诸多使用场景，深度学习库必须注重设计的灵活性与功能的通用性，才能在风起云涌的人工智能生态系统中得以立足。TensorFlow 作为当前主流的深度学习库之一，其设计具有很高的灵活性和通用性，主要体现在以下几个方面。

- 在算子定义方面，相比于其他深度学习库，TensorFlow 提供的算子粒度更细、数量更多，能够支撑上层灵活多变的模型和算法。用户可以使用这些算子自由、灵活地开发各种深度学习模型。此外，很多传统的机器学习模型也可以基于 TensorFlow 实现，如支持向量机、决策树和随机森林等。TensorFlow 亦支持深度学习和传统机器学习混合的模型，从而使得数据流水线式的应用创新成为可能。TensorFlow 对新算子的支持也足够灵活，允许用户通过组合已有的细粒度算子来构造新的算子，以快速实现算法原型、验证一些新的想法。用户也可以使用 C++ 语言和 CUDA 等底层函数库实现新的算子并在运行时动态加载使用，以便满足专用算法需求并保证计算性能。
- 在编程范式方面，TensorFlow 支持声明式编程，将模型的定义和执行解耦。模型以数据流图结构表示，经过编译和优化之后才会真正执行。以数据流图抽象为核心的设计在保证模型执行效率的同时，使得用户编程更加灵活。例如，在模型定义阶段，用户可以通过添加控制依赖边来指定算子的执行顺序，通过添加自定义变量自如地管理数据流图的输入和输出，还可以通过队列控制多设备之间的数据传输和子图执行时序。在数据流图的运行态，用户可以指定数据流图中待执行的子图，从而避免不必要的算子计算开销。除了模型之外，数据读取、数据预处理等其他操作都可以被添加到数据流图中，用户可以通过编辑数据流图实现对具体应用的端到端灵活控制。
- 在运行时框架方面，TensorFlow 在具备隐式并行计算能力的同时，也提供了细粒度的显式控制接口，允许灵活地控制模型在多节点异构设备上的分布式执行方式。用户在编写深度学习模型时，可以自由地将模型中的每个算子绑定在任意的计算设备上。TensorFlow 运行时框架负责将模型对应的数据流图按照设备进行切分，并自动插入必要的通信操作，对用户屏蔽了底层复杂的数据传输与时序同步机制。用户可以结合具体模型和应用

的特点，通过手工指定或者以强化学习等方式找出模型在多节点异构硬件上的最优布局与执行方式（如数据并行、模型并行等），这使得模型的训练和推理都有更多的优化空间。

- 在多语言支持方面，TensorFlow 提供 Python、C、C++、Java、Go 等主流语言的编程接口。虽然 Python 是当前深度学习和人工智能领域使用最为广泛的编程语言，但是其他语言也有各自的语法优势、适用场景以及拥趸，它们能够满足科研、商用等不同应用领域及服务器、终端等不同目标设备的开发需求。另外，社区开源贡献者在 TensorFlow C API 基础上扩展开发了 Node.js、Julia、R 等其他语言的编程接口，这在体现 TensorFlow 内核设计灵活性的同时，也进一步扩大了其作为通用深度学习库的场景覆盖范围。

总而言之，TensorFlow 通过丰富的算子、灵活的编程范式、自由的运行时框架以及多语言 API 支持，对用户展现了高度的灵活性和通用性。TensorFlow 在其设计之初就被定义为灵活通用的深度学习平台。Google 公司将它开源的目的正是要以其作为基石，构筑深度学习乃至整个人工智能的生态圈。

1.2.2 端云结合的人工智能引擎

“端云结合”是当今信息化、智能化技术发展的普遍趋势。一方面，随着信息技术在生产、生活中各个应用领域愈发深入的集成，对海量数据的高效处理成为 IT 服务商与决策部门的迫切需求。在传统数据中心基础上发展起来的云计算和大数据技术以集约化的资源管理、动态弹性的资源供给为持续膨胀的应用提供了高水平、可伸缩的计算能力，同时降低了服务提供者的准入门槛。这就要求传统服务器端软件必须适应云化部署场景，以水平扩展（scale-out）、无状态、微服务等方式构建高内聚、低耦合的系统架构。另一方面，随着以智能手机为代表的移动终端技术的高速普及，以及物联网、机器人等智能化技术在传统行业的不断渗透，用户对于数据私密性、安全性的重视程度逐渐增强，应用场景对服务实时性与可用性的要求也更加严格。在这一背景下，计算能力的边缘化成为与云化并驾齐驱的演进方向。这就要求提供服务的软件能够适应体系结构多样、计算资源有限、功耗受到制约的终端硬件环境，并具备一定的自治与协同工作能力。

在人工智能领域野心勃勃的 Google 公司自然准确地把握并积极地引领着这一趋势。在云侧，Google 既是一家公有云提供商，需要通过具有核心竞争力的 PaaS 层产品吸引企业级用户和二次开发者；又是一家业务高度依赖于智能算法与海量数据的创新型公司，需要开发高效、灵活的基础平台软件满足自身业务快速发展的需要。在端侧，Google 公司不但需要为日新月异的应用开发提供强有力的智能化支撑能力，从而使其 Android 生态系统得以抗衡强大的竞争对手；而且需要借助实时且可靠的智能计算引擎进军物联网、可穿戴设备、增强实现、无人驾驶等新兴领域，以寻求“后移动互联网时代”的新增长点。TensorFlow 等平台层软件的设计因此也兼顾了云侧与端侧的需求。

TensorFlow 对云计算场景的支持是其竞争力的基础，主要体现在以下方面。

- 提供多种标准化的安装包、构建脚本及容器化封装，支持在不同 Linux 发行版以及 Windows Server 等其他服务器操作系统上部署，既允许以二进制包方式快速安装，也允许针对特定环

境定制化编译高效的目标代码，从而极大地扩展了软件的适用范围，增强了适配能力。

- 支持对接多种常见的公有云和私有云服务，如 Google Cloud Storage、Amazon S3、HDFS，并为对接其他类似服务预留可扩展设计，从而能够与既有的互联网、大数据生态系统无缝交互，实现资源复用与服务组合。
- 兼容若干种常见的高性能计算与通信硬件，能够有效利用云环境的既有投资并提升应用软件对高端硬件资源的利用率。例如，支持 NVIDIA 和 OpenCL GPU，能够充分挖掘众核设备的并行计算能力；支持 RDMA 网络协议，能够充分发挥 InfiniBand 等高速网络设备的带宽潜力。
- 灵活的运行时框架设计，既提供标准且易用的 PS-worker 分布式模式，也允许用户自由开发针对特定环境需求的分布式框架。即使脱离 Google 公司的 Borg 等基础设施，以 TensorFlowOnSpark、MaTEx-TensorFlow 为代表的第三方工具也能够利用既有的分布式平台提升 TensorFlow 在数据中心和超算集群中的可伸缩性。

TensorFlow 在端侧场景方面也毫不逊色，其主要设计体现在以下几个方面。

- 推理（预测）态代码能够运行于多种主流的终端平台，包括 Android、iOS，以及部署 Linux 操作系统的多类 ARM 与 MIPS 设备（如 Raspberry Pi），从而为形态多样的终端设备集成 AI 认知与决策能力提供支撑。
- 通过 XLA AOT（ahead-of-time）编译技术及其他软硬件解耦设计，显著地简化底层异构计算设备的对接方式，实现对神经网络芯片等新型专用端侧硬件的快速支持能力。
- 提供量化参数和低精度代数等算法层机制，适配算力、存储和功耗受限的终端，从而实现低端边缘设备的智能化。
- 提供模型与框架一体化的精简版运行时平台，具备完全的离线工作能力，有助于实现端侧计算的私密性与实时性。

可以看出，TensorFlow 作为一套人工智能引擎，不但致力于增强应用系统的“大脑”，同时也在帮助其完善“末梢神经”。可以预见，未来 TensorFlow 等人工智能引擎会像 Linux 操作系统内核一样，成为在端云两侧广泛支撑各类应用、助力实现智能化社会的幕后英雄。

1.2.3　高性能的基础平台软件

虽然如今的互联网、大数据计算平台软件层出不穷，核心技术变幻莫测，吸睛特性轮番登场，但是性能始终都能够超脱于名目繁多的噱头，成为几乎所有用户一致认可的硬指标。在半导体器件物理极限将至的情况下，摩尔定律的有效性已经存疑，软件轻松分享硬件发展红利的时代走向末路。硬件设计者正在广泛采纳新型器件、三维电路、应用定制、众核并行等多元化思路满足应用不断增长的算力需求，这为软件开发者提供机会的同时也带来了不小的挑战。如何将软件架构和算法有效适配到硬件体系结构、充分利用硬件资源发挥其设计性能，成为所有软件开发者，特别是基础平台层软件开发者面临的重要问题。

随着深度学习技术发展而兴起的一系列开源计算库长期处于激烈竞争的态势。在竞争过程中，一快遮百丑。这里的“快”字有两层含义：深度学习库的开发者不仅需要快速响应上层需求和下层技术的变化，及时发布新版本与新特性，而且需要通过苦修内功提升深度学习库本身的性能，加快算法模型的训练与推理速度。在第一个“快”字上，TensorFlow 借助 Google 公司强大的号召力和坚决的执行力，长期保持领先地位。在第二个“快”字上，TensorFlow 曾因权衡灵活性等原因一度落后于同类软件，但如今已经迎头赶上，在主流应用场景中取得了优异的测试成绩。这归功于众多核心研究者和开源贡献者在性能方面的深耕。

TensorFlow 的高性能设计首先体现在它对高端和专用硬件的深入支持。同其他主流的深度学习库一样，TensorFlow 将 NVIDIA GPU 作为训练态的硬件加速器，同时兼顾 OpenCL GPU 设备。不同于简单使用 CUDA Runtime API 的其他平台，Google 的工程师基于 CUDA Driver API 实现了控制粒度更细、并行性能更优的 StreamExecutor 异构计算库，并对 cuBLAS、cuDNN 等库的函数变种进行了精确的适配。在推理态，尽管 Google 没有开源或销售 TPU，然而 TensorFlow 开放性的设计已经促使多家芯片厂家实现了对接，这为定制化设备上的计算性能提升提供了保障。针对高性能计算环境中常用的 InfiniBand、RoCE 等高速网络设备，以及 NVLink 等片间高速互联技术，TensorFlow 引入了 RDMA、NCCL 等协议，较好地解决了通信延迟问题，推进了分布式计算作业的加速比提升。

其次，系统层的优化技术是 TensorFlow 性能提升的重要杀手锏。相比来自于学术界的算法研究团队，Google 科研与工程团队深厚的系统研发背景是 TensorFlow 构建性能竞争力的坚实后盾。XLA 这种融合了编译器设计理论的优化框架就是一例。它引入的 JIT（just-in-time）编译机制能够在数据流图运行过程中实时创建二进制代码，将其中大量细粒度的操作融合为少量粗粒度的专用核函数，从而减少图中操作执行时的内存分配和上下文切换开销，极大地提升计算速度。TensorFlow 诸多模块设计中也存在着细节上的性能优化。例如，通信模块中具有若干种旁路（bypass）设计，可以避免不必要的网络访问和内存复制开销；数据流图构建时会执行常量折叠、公共子表达式消除、内联函数展开等多种语法树优化，能够消除无意义的计算开销。这些设计体现出开发者良好的软件工程素养与精益求精追求。

最后，算法层的优化设计也是 TensorFlow 实现优异性能不可或缺的组成部分。为了实现高性能的目标，TensorFlow 的设计采纳了自顶向下、全栈优化的思路，而算子恰恰是贯穿上下层的核心要素。在深度学习的算法模型中，每种算子的逻辑都可以采用多种算法实现。为此，TensorFlow 内置了多种优化后的基础算子和模型组件。以卷积算子为例，cuDNN 提供了 Winograd 等 8 种算法。针对不同的输入数据大小、卷积计算超参以及内存等资源限制，TensorFlow 会自动为每个卷积操作选择最快的实现算法。另外，针对递归神经网络等模型，TensorFlow 也支持 Fold 解决方案，使得动态批处理成为可能，极大提高了这些模型的计算速度。

综上所述，性能是 TensorFlow 研发者重点关注的设计目标。虽然 TensorFlow 开源版本的性能优化起步稍晚，但是在 Google 团队和开源社区的共同努力下进步迅速。在这个“天下武功唯快不破”的时代，TensorFlow 的高性能优势必将为 Google 插上腾飞的翅膀，使之引领人工智能研究与应用的高速发展。

1.3　基本架构

在展开介绍 TensorFlow 的使用方法和设计原理之前，我们有必要建立对 TensorFlow 基本架构的直观认识。本节从工作形态和组件结构这两个角度对 TensorFlow 进行概要性的说明。读者可以以此为切入点，逐步理顺学习 TensorFlow 的脉络。

1.3.1　工作形态

基础平台层软件的设计模式多种多样，它们对应用层开发者体现出的工作形态也有所差别。在众多平台设计模式中，存在两类基础而典型的模式，即图 1-2 所示的库模式和框架模式。在库模式下，平台层软件以静态或动态的开发库（如 .a、.so 文件）形式存在，应用层开发者需要编写程序调用这些库提供的函数，实现计算逻辑。程序的入口（如 `main()` 函数）及整体流程控制权把握在应用层开发者手中。在框架模式下，平台层软件以可执行文件的形式存在，并以前端交互式程序或后端守护进程方式独立运行。应用层开发者需要遵从平台规定的接口约束，开发包含计算逻辑在内的子程序，交由框架性质的平台层软件调度执行。程序的入口及整体流程控制权由框架把握。

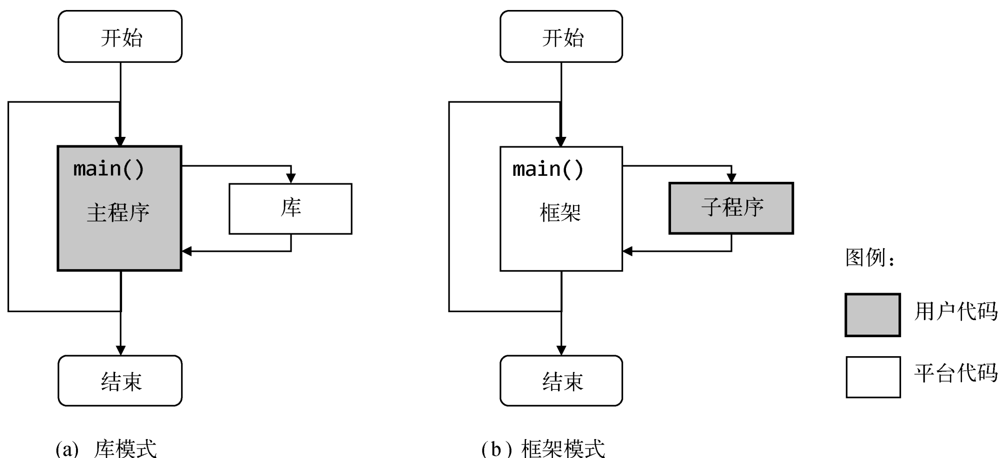


图 1-2　平台层软件的典型设计模式

在高性能与大数据计算领域，典型的库模式软件有用于计算的 Eigen、NumPy，以及用于通信的 MPI、ZeroMQ 等。基于这些库开发应用时，编程方式比较灵活，部署模式也相对轻量。应用开发者具有较大的自由度，但不得不编写业务逻辑之外的不少“脚手架”代码，以便将算法代码片段转变为完整可用的软件。典型的框架模式软件有大数据计算平台 Hadoop、Spark，以及基于 SQL 和类 SQL 语言的数据库、数据仓库等。使用这些框架开发应用时，开发者的工作相对轻松，只需要编写与业务逻辑密切相关的算法代码，不用关心运行时机制的复杂性。不过，程序的灵活性将受制于框架的约束。

TensorFlow 的设计采用了库模式。之所以如此，是出于灵活通用、端云结合及高性能等设计目标的考虑。库模式的平台层软件便于与各种既有的框架协同工作，不对软件的运行时组件添加新的约束，应用范围也不受制约。除了依赖最基本的编程语言库和操作系统调用，这类平台层软件同其他环境因素解耦，从而可以做到高度的可移植性。在单机和终端等场景下，由于没有守护进程和调度框架的开销，有效计算逻辑的资源利用率也会提高，进而有助于性能优化。

综上，TensorFlow 的工作形态是由用户编写主程序代码，调用 Python 或其他语言函数库提供的接口以实现计算逻辑。用户部署和使用 TensorFlow 系统时，不需要启动专门的守护进程，也不需要调用特殊的启动工具，只需要像编写普通的本地应用程序那样即可上手。用户也不用担心库模式的开发所必需的那些“脚手架”代码，因为 TensorFlow 已经提供了多种高级抽象，尽可能地最小化了核心计算逻辑之外的开发工作。

1.3.2 组件结构

TensorFlow 作为一套包含数十万行代码的大型软件，其组件结构较为复杂。不过，由于其代码组织合理，文档资料充分，我们很容易将它的软件结构进行不同抽象程度的宏观呈现。初识 TensorFlow 的新手只需要从最高层的抽象视角观察其组件构成。图 1-3 给出了一幅粗粒度的 TensorFlow 组件结构示意图，展示了 TensorFlow 的主要内部结构及其与周边环境的关系。

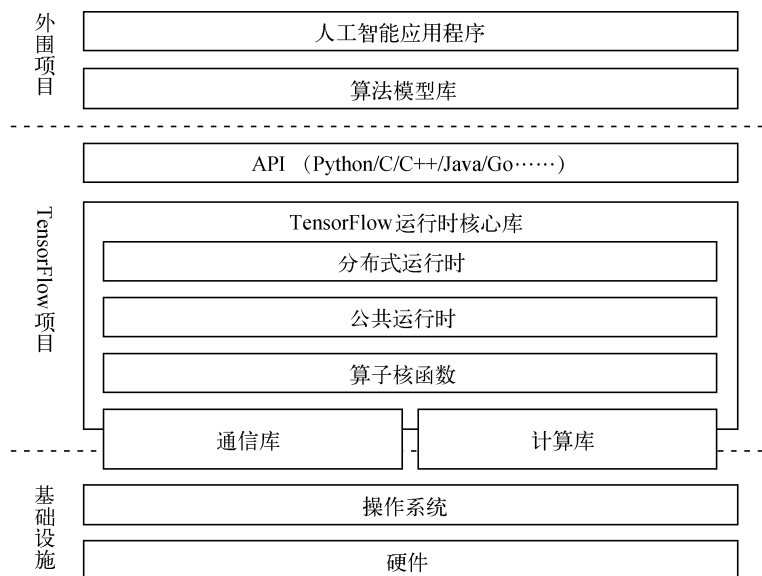


图 1-3 TensorFlow 的组件结构示意图

构成 TensorFlow 的主体是其运行时核心库。对于普通的 Python 应用层开发者而言，这个核心库就是指通过 `pip` 命令等方式安装 TensorFlow 之后，部署到 site-packages 或类似目录中的动态链接库文件。生成这个库的 C++ 源代码大致分为 3 个层次：分布式运行时、公共运行时和算子

核函数。其中，公共运行时实现了数据流图计算的基本逻辑，分布式运行时在此基础上实现了数据流图的跨进程协同计算逻辑，算子核函数则包含图上具体操作节点的算法实现代码。

TensorFlow 运行时核心库导出的函数接口基于 C 和 C++ 语言。为了使用其他语言进行应用开发，TensorFlow 提供了多语言的 API 层。Python 应用层开发者在代码中调用 `import tensorflow as tf` 时，导入的便是 TensorFlow 安装在 Python 第三方库目录下的 API 层模块（本书后面沿用这种 Python 包导入惯例，使用 `tf` 作为 `tensorflow` 命名空间的缩写）。API 层对用户屏蔽了 TensorFlow 核心库的动态链接逻辑，使得用户可以使用自己熟悉的语言编写算法模型。

为了简化经典模型的开发，使得 TensorFlow 成为一套“开箱即用”的工具，Google 官方团队及开源贡献者们在 TensorFlow 社区开设了若干算法模型库及人工智能应用程序项目。用户可以复用这些项目的成果，加快自己的项目开发进度；也可以学习它们的实现原理，提升自己的模型与应用设计水平。这些外围项目中的部分代码（如 Keras）已被认为具有较高的共性价值，因此逐步被加入到 TensorFlow 主项目之中。

TensorFlow 运行时核心库底层对接的是各种计算库和通信库。这些库有的是外部组件（如用于 CPU 代数计算的 Eigen 库），有的则作为 TensorFlow 源代码的一部分集成在核心库内部（如用于 GPU 并行计算的 StreamExecutor 库）。用户在开发应用程序时看不到这些库的细节，只需要按照软件文档安装好必要的外部依赖包即可。

上面所有组件均运行在本地操作系统和硬件基础设施之上。在服务器端运行场景，最常见的宿主操作系统是 Linux，硬件一般为 x86 CPU 和 NVIDIA GPU。在移动终端运行场景，宿主操作系统可以是 Android、iOS 等，硬件一般为 ARM CPU 和专用的人工智能芯片。TensorFlow 不仅支持原生的物理环境，对虚拟机和容器也完全兼容，这构成了云计算环境下的最佳实践。

1.4　小结

作为一套优秀的深度学习计算库，TensorFlow 在新一波人工智能浪潮中脱颖而出。它源于 Google 公司内部基于海量数据开展感知和预测类应用的需求，并通过围棋大战向公众一展雄姿。依托 Google 团队雄厚的科研实力，同时借助开源社区的集体智慧，TensorFlow 已经成为了一套运算性能强劲、框架设计通用、语言接口丰富，并支持生产环境部署和端云协同计算的通用人工智能基础平台软件。TensorFlow 的构架设计灵活而开放，有助于适应多样的应用场景并吸引第三方开发者贡献特性。我们相信，在当今人工智能理论进步与应用落地并举的时代，TensorFlow 势必会引领相关研究与工程领域的高速发展。

第 2 章 TensorFlow 环境准备

对 TensorFlow 有了基本认识之后，读者也许会迫不及待地想在自己的计算机上部署一套 TensorFlow 开发和运行环境。安装 TensorFlow 并对其软件整体结构与关键组件有所了解是学习 TensorFlow 基础用法和算法模型开发方法的前提。本章首先向读者介绍在 Linux 操作系统上安装 TensorFlow 的多种方法，然后介绍 TensorFlow 的主要软件依赖项，以及 TensorFlow 的源代码和安装目录组织结构。在指导读者构建 TensorFlow 开发和运行环境的同时，我们构建了一幅学习 TensorFlow 的宏观视图。

2.1 安装

TensorFlow 提供多种语言的 API，具体包括 Python、Java、Go、C 和 C++ 等。本节重点介绍 TensorFlow Python 发布包的安装和开发环境搭建方法。选择 Python 的原因有二：其一，Python 是目前机器学习和深度学习领域使用最为广泛和最受欢迎的程序设计语言；其二，Python 语言本身简单易学，适合新手快速入门。TensorFlow 团队为 Linux、macOS 和 Windows 这 3 大主流操作系统分别适配了 CPU 和 GPU 版本的 Python 发布包（从 TensorFlow 1.2 开始，TensorFlow 团队不再为 macOS 提供 GPU 版本的官方支持）。本节将详细介绍 TensorFlow 的多种安装方法，具体包括使用 Anaconda、原生 pip 和 virtualenv 安装，以及使用 Docker 容器安装和从源代码编译安装。本节以 Linux 发行版——Ubuntu 为例进行说明和展示。

2.1.1 TensorFlow安装概述

TensorFlow Python 发布包支持使用多种 Python 包管理工具进行安装。常见的 Python 包管理工具有 Anaconda、原生 pip 和 virtualenv。同时，用户也可以自己下载 TensorFlow 的源代码进行编译，然后得到对应的 whl 包。whl 是 Python 软件发布的新格式，用于取代曾经风靡一时的 egg 包格式。除此以外，我们还可以使用 Docker 运行包含 TensorFlow 二进制包的容器，实现在容器中使用 TensorFlow。到目前为止，TensorFlow 官方主要提供了以上 5 种安装方法。根据它们各自在 Linux、macOS 和 Windows 上的支持情况，我们整理出如表 2-1 所示的 TensorFlow Python 发布包的安装适配情况。为了简化说明，后面的所有 TensorFlow 发布包均特指 TensorFlow Python 发布包。

表 2-1　TensorFlow 安装适配表

	Anaconda	原生 pip	virtualenv	Docker	源代码
Linux	支持	支持	支持	支持	支持
macOS	支持	支持	支持	支持	支持
Windows	支持	支持	支持	不支持	实验性支持

在加速器硬件方面，TensorFlow 官方目前主要支持 NVIDIA 公司的 GPU 设备（后面的 GPU 特指 NVIDIA GPU）。如果用户想要使用 GPU 版的 TensorFlow，那么必须安装相应的显卡驱动程序和配套软件，具体包括统一计算工具包 CUDA、深度神经网络 GPU 加速库 cuDNN（CUDA® Deep Neural Network library）和 CUDA 分析工具接口——libcupti-dev 开发库。其中，CUDA 囊括了 NVIDIA 显卡驱动程序和基于 GPU 的通用计算程序，我们将在 2.2.4 节中详细介绍它。cuDNN 是 NVIDIA 为深度学习设计的 GPU 加速库，它对常见的深度神经网络都做了针对性的优化。随着 GPU 技术的不断发展，CUDA 和 cuDNN 也在不断更新，表 2-2 给出了 TensorFlow 各版本对 CUDA 和 cuDNN 的官方支持情况。

表 2-2　TensorFlow 各版本对 CUDA 和 cuDNN 的支持情况（最低版本要求）

TensorFlow	CUDA	cuDNN
0.8 ~ 0.11	7.0	4.0
0.12 ~ 1.2	8.0	5.1
1.3 ~ 1.4	8.0	6.0

因为 CUDA 的开发环境对系统软硬件存在较强依赖和约束，所以读者应该检查自己的机器是否满足以下 3 项指标：

- ❑ GPU 是否支持 CUDA；
- ❑ 系统的编译器和工具链是否支持 CUDA；
- ❑ 系统内核是否满足特定 CUDA 版本的安装要求。

根据 NVIDIA 官网的介绍，我们整理出 x86_64 架构下，CUDA 8.0 当前支持的 Linux 发行版以及对应的内核版本和编译器，如表 2-3 所示。

表 2-3　CUDA 8.0 支持的 Linux 发行版

Linux 发行版	系统内核版本	GCC	glibc	ICC	PGI	Clang
RHEL 7.x	3.10	4.8.2	2.17	15/16	16.3+	3.8+
RHEL 6.x	2.6.32	4.4.7	2.12	15/16	16.3+	3.8+
CentOS 7.x	3.10	4.8.2	2.17	15/16	16.3+	3.8+
CentOS 6.x	2.6.32	4.4.7	2.12	15/16	16.3+	3.8+
Fedora 23	4.2.3	5.3.1	2.22	15/16	16.3+	3.8+
openSUSE 13.2	3.16.6	4.8.3	2.19	15/16	16.3+	3.8+
SLES 12	3.12.28	4.8.6	2.19	15/16	16.3+	3.8+
SLES 11 SP4	3.0.101	4.3.4	2.11	15/16	16.3+	3.8+
Ubuntu 16.04	4.4.0	5.3.1	2.23	15/16	16.3+	3.8+
Ubuntu 14.04	3.13	4.8.2	2.19	15/16	16.3+	3.8+

关于 CUDA 8.0 和 cuDNN 5.1 的安装和配置方法，读者可以参考 NVIDIA 官网的详细步骤说明（http://docs.nvidia.com/cuda/cuda-installation-guide-linux），这里不再赘述。下面我们介绍一下 TensorFlow 开发环境的安装方法。

2.1.2 使用Anaconda安装

Anaconda 是目前很受欢迎的 Python 数据科学平台，它内置超过 1000 个 Python 数据科学软件包，并同时提供社区版和官方发布版的软件包。作为 Python 软件包管理工具，它也同时支持 Python 2.7 和 Python 3.6 的软件包管理。Anaconda 在主流的操作系统上都能正常运行，如 Linux、macOS 和 Windows。读者可以在 Anaconda 官网上（https://www.continuum.io/downloads）查看和下载相应的版本。

在使用 Python 做开发时，时常出现依赖软件包版本冲突的情况，而 Anaconda 提供的虚拟开发环境有效解决了该问题。它支持为项目创建特定的虚拟环境，允许每个虚拟环境拥有一套独立的 Python 软件包。于是，我们就可以在不同的虚拟环境中开发可能存在冲突的 Python 项目。

使用 Anaconda 安装 TensorFlow 主要分为以下 4 个步骤。

(1) 下载并安装 Anaconda。安装结束前，脚本会询问我们是否将 Anaconda 路径添加到 $PATH 环境变量，并将更新的 $PATH 保存到 /home/<user>/.bashrc 文件中。这里建议选择 Yes。这样一来，再次打开终端时，shell 程序便能自动加载 /home/<user>/.bashrc 文件中更新的 $PATH。此外，用户亦可使用 `source` 命令，在当前 shell 环境中引入该环境变量：

```
$ bash Anaconda2-4.4.0.1-Linux-x86_64.sh
$ source ~/.bashrc
```

(2) 使用 Anaconda 的命令行工具 `conda` 创建虚拟开发环境。其中`-n` 参数表示环境名称，我们将其命名为 `tensorflow`。同时，我们指定虚拟环境使用的 Python 版本为 2.7：

```
$ conda create -n tensorflow python=2.7
```

(3) 使用 `source activate` 命令激活并进入 Anaconda 创建的虚拟开发环境：

```
$ source activate tensorflow
(tensorflow)$  # 进入 tensorflow 虚拟开发环境
```

(4) 在虚拟开发环境中安装 TensorFlow，这里以 CPU 版的 TensorFlow 1.2.1 为例：

```
(tensorflow)$ pip install \
https://storage.googleapis.com/tensorflow/linux/cpu/ \
tensorflow-1.2.1-cp34-cp34m-linux_x86_64.whl
```

至此，TensorFlow 安装完成。关于如何验证 TensorFlow 是否安装成功，我们将在 2.1.7 节中统一介绍。下面我们继续介绍如何使用原生 pip 安装 TensorFlow。

2.1.3 使用原生pip安装

原生 pip 是 Python 社区推荐的软件包安装和维护工具。针对 Python 2.*x* 和 Python 3.*x* 版本，

该工具分别提供 `pip` 和 `pip3` 命令。PyPA（Python Packaging Authority）工作组维护着 pip 的软件包下载源。在开始使用原生 pip 安装 TensorFlow 前，我们应该首先检查本地安装的 Python 版本：

```
$ python -v
Python 2.7.12
```

这里强烈推荐使用 pip 8.1 以上版本安装 TensorFlow。更老版本的 pip 存在一些 bug，有可能影响 TensorFlow 的安装和使用。下面我们以 Ubuntu 系统为例，说明 Python 2.*x* 和 Python 3.*x* 升级 pip 版本的方法：

```
$ sudo apt-get install python-pip python-dev    # 用于 Python 2.x
$ sudo apt-get install python3-pip python3-dev  # 用于 Python 3.x
```

因为 TensorFlow 已经加入了 PyPA 维护的软件包集合，所以可以使用原生 pip 直接安装 TensorFlow。针对不同 Python 版本和计算设备的安装命令如下所示，用户可以依据自己的环境选择其一：

```
$ pip install tensorflow      # 用于 Python 2.x，CPU 版（不支持 GPU）
$ pip3 install tensorflow     # 用于 Python 3.x，CPU 版（不支持 GPU）
$ pip install tensorflow-gpu  # 用于 Python 2.x，GPU 版（支持 CPU）
$ pip3 install tensorflow-gpu # 用于 Python 3.x，GPU 版（支持 CPU）
```

命令执行后，TensorFlow 软件包将被安装到操作系统全局或者当前用户本地的 Python 库目录（这取决于使用 root 用户还是一般用户安装）。事实上，原生 pip 安装方式与 Anaconda 安装方式并不冲突，二者可以配合使用。因此，用户也可以在 Anaconda 虚拟环境中执行上述命令，以便将 TensorFlow 安装到虚拟环境。

2.1.4　使用virtualenv安装

virtualenv 与 Anaconda 类似，也是一种 Python 软件包管理工具，同样支持创建虚拟开发环境。使用 virtualenv 安装 TensorFlow 的步骤简述如下。

(1) 安装 pip 和 virtualenv：

```
$ sudo apt-get install python-pip python-dev python-virtualenv # 用于 Python 2.x
$ sudo apt-get install python3-pip python3-dev python-virtualenv # 用于 Python 3.x
```

(2) 使用 virtualenv 创建 Python 虚拟开发环境。`virtualenv` 命令的最后一个参数为虚拟环境的目录名：

```
$ virtualenv --system-site-packages ~/tensorflow # 用于 Python 2.x
$ virtualenv --system-site-packages -p python3 ~/tensorflow # 用于 Python 3.x
```

(3) 激活并进入虚拟开发环境：

```
$ source ~/tensorflow/bin/activate
(tensorflow)$  # 进入 tensorflow 虚拟开发环境
```

(4) 在虚拟开发环境中安装 TensorFlow：

```
(tensorflow)$ pip install tensorflow      # 用于 Python 2.x，CPU 版（不支持 GPU）
(tensorflow)$ pip3 install tensorflow     # 用于 Python 3.x，CPU 版（不支持 GPU）
(tensorflow)$ pip install tensorflow-gpu  # 用于 Python 2.x，GPU 版（支持 CPU）
(tensorflow)$ pip3 install tensorflow-gpu # 用于 Python 3.x，GPU 版（支持 CPU）
```

2.1.5 使用Docker安装

Docker 是目前最为流行的 Linux 容器技术，它分为社区版（Community Edition，CE）和企业版（Enterprise Edition，EE）。前者主要为个人开发者和小型研发团队提供设计容器应用的基本开发环境，后者主要为企业开发者提供更加安全和可靠的商用生产环境。Docker 官网的文档（https://docs.docker.com/engine/installation/）介绍了在各种操作系统上安装 Docker 的详细步骤。现在，我们假设读者的机器上已经安装好 Docker。下面的代码展示了使用 Docker 运行 TensorFlow 容器应用的命令格式和常用输入参数：

```
$ docker run -it -p hostPort:containerPort TensorFlowCPUImage
```

其中，`hostPort:containerPort` 是可选参数。`hostPort` 表示容器映射到宿主机的端口号，`containerPort` 表示容器应用的端口号。如果读者想要在命令行中使用 TensorFlow，那么无需输入此参数。如果读者想要在 Jupyter Notebook 中使用 TensorFlow，那么需要将 `hostPort` 和 `containerPort` 设置为相同的端口，如 8888。`TensorFlowCPUImage` 是指使用 TensorFlow CPU 版镜像。Google 为 TensorFlow CPU 版创建了 4 种不同的镜像文件，并保存在 Google 的镜像仓库中，具体如下所示。

- gcr.io/tensorflow/tensorflow：二进制镜像。
- gcr.io/tensorflow/tensorflow:latest-devel：二进制和源代码镜像。
- gcr.io/tensorflow/tensorflow:*version*：指定版本的二进制镜像。
- gcr.io/tensorflow/tensorflow:*version*-devel：指定版本的二进制和源代码镜像。

我们使用 TensorFlow CPU 版二进制镜像，并在启动容器时设置相应的端口号。下面的命令成功运行后，会输出 Jupyter Notebook 服务运行的地址，本例中便是本地 IP 的 8888 端口对应的地址（http://localhost:8888）：

```
$ docker run -it -p 8888:8888 gcr.io/tensorflow/tensorflow
```

如果想要使用 GPU 版本的 TensorFlow 容器应用，那么还需要安装 NVIDIA 开发的 nvidia-docker（https://github.com/NVIDIA/nvidia-docker）工具。下面的代码展示了如何在 Ubuntu 上安装 nvidia-docker：

```
$ wget -P /tmp \
https://github.com/NVIDIA/nvidia-docker/releases/download/v1.0.1/ \
nvidia-docker_1.0.1-1_amd64.deb
$ sudo dpkg -i /tmp/nvidia-docker*.deb && rm /tmp/nvidia-docker*.deb

# 安装完成后，测试 nvidia-smi 是否可用
$ nvidia-docker run --rm nvidia/cuda nvidia-smi
```

TensorFlow GPU 版的镜像文件同样分为 4 种，如下所示。

- gcr.io/tensorflow/tensorflow:latest-gpu：二进制镜像。
- gcr.io/tensorflow/tensorflow:latest-devel-gpu：二进制和源代码镜像。
- gcr.io/tensorflow/tensorflow:*version*-gpu：指定版本的二进制镜像。
- gcr.io/tensorflow/tensorflow:*version*-devel-gpu：指定版本的二进制和源代码镜像。

启动 TensorFlow GPU 版容器应用的输入参数与 CPU 版类似，只是启动命令需要换为 nvidia-docker，如下所示：

```
$ nvidia-docker run -it -p 8888:8888 gcr.io/tensorflow/tensorflow:latest-gpu
```

下面我们介绍如何使用源代码编译安装 TensorFlow。

2.1.6　使用源代码编译安装

为了保证软件对操作系统和硬件平台的通用性，Google 官方发布的 TensorFlow whl 包没有使用过多的编译优化选项，如 XLA、AVX、SSE 等。如果读者想要打开这些编译优化选项来提升 TensorFlow 的计算性能，那么必须使用源代码编译安装的方式。TensorFlow 官方支持在 Linux 和 macOS 操作系统上进行源代码编译。目前，对于 Windows，只是提供了实验性的源代码编译方案。

TensorFlow 源代码编译的目标输出是 TensorFlow whl 包，构建工具是 Google 开源的一套软件构建工具——Bazel。在开始源代码编译前，用户应该确保开发环境上已经安装好 Bazel 以及 TensorFlow 依赖的第三方 Python 软件包。如果想要编译 GPU 版的 TensorFlow whl 软件包，还需要安装配套的 NVIDIA CUDA 和 cuDNN。下面我们仍然以 Ubuntu 为例，介绍源代码编译 TensorFlow 的流程。

(1) 安装 Bazel 软件构建工具（在其他操作系统上安装 Bazel 的方法可参考其官方网站的文档）：

```
# 1.安装 Bazel 依赖的 JDK 8
$ sudo apt-get install openjdk-8-jdk
# 2.添加 Bazel 发布包 URL 到 apt 的下载源
$ echo "deb [arch=amd64] http://storage.googleapis.com/bazel-apt stable jdk1.8" | \
sudo tee /etc/apt/sources.list.d/bazel.list
$ curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -
# 3.使用 apt 安装 Bazel
$ sudo apt-get update && sudo apt-get install bazel
# 4.（可选步骤）当 Bazel 版本较低时，使用 apt 更新 Bazel
$ sudo apt-get upgrade bazel
```

(2) 安装 TensorFlow 依赖的第三方 Python 软件包：

```
$ sudo apt-get install python-numpy python-dev python-pip python-wheel # 用于 Python 2.x
$ sudo apt-get install python3-numpy python3-dev python3-pip python3-wheel # 用于 Python 3.x
```

这里主要包括下面 4 个包。

- numpy：经典的数据处理软件包。
- dev：Python 扩展开发工具包。
- pip：安装和管理 Python 软件包的工具包。
- wheel：Python 软件打包和发布工具包。

(3)（可选步骤）安装适配当前操作系统的 NVIDIA CUDA 和 cuDNN，以及 CUDA 分析工具接口 libcupti-dev 开发库。

(4) 从 GitHub 上克隆 TensorFlow 源代码，并将其切换到稳定的发布分支，如 TensorFlow 1.2：

```
$ git clone https://github.com/tensorflow/tensorflow.git
$ cd tensorflow
$ git checkout r1.2
```


(5) 运行 configure 脚本，配置 TensorFlow 编译选项。该脚本位于 TensorFlow 项目源代码的根目录。下面的命令行输出展示了用户可配置的编译选项，包括 Python 路径、是否支持 Google 云平台、是否支持 HDFS、是否使用 XLA 编译优化、是否使用 OpenCL、是否使用 CUDA 等。对于期望开启的选项，用户可以根据提示，输入“y”或相应软件安装目录：

```
$ ./configure
Please specify the location of python. [Default is /usr/bin/python]: /usr/bin/python2.7
Please specify optimization flags to use during compilation when bazel option "--config=opt"
is specified [Default is -march=native]:
Do you wish to use jemalloc as the malloc implementation? [Y/n]
jemalloc enabled
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N]
No Google Cloud Platform support will be enabled for TensorFlow
Do you wish to build TensorFlow with Hadoop File System support? [y/N]
No Hadoop File System support will be enabled for TensorFlow
Do you wish to build TensorFlow with the XLA just-in-time compiler (experimental)? [y/N]
No XLA JIT support will be enabled for TensorFlow
Found possible Python library paths:
  /usr/local/lib/python2.7/dist-packages
  /usr/lib/python2.7/dist-packages
Please input the desired Python library path to use.  Default is
[/usr/local/lib/python2.7/dist-packages]
Using python library path: /usr/local/lib/python2.7/dist-packages
Do you wish to build TensorFlow with OpenCL support? [y/N]
No OpenCL support will be enabled for TensorFlow
Do you wish to build TensorFlow with CUDA support? [y/N] Y
CUDA support will be enabled for TensorFlow
Please specify which gcc should be used by nvcc as the host compiler. [Default is /usr/bin/gcc]:
Please specify the Cuda SDK version you want to use, e.g. 7.0. [Leave empty to use system default]:
8.0
Please specify the location where CUDA 8.0 toolkit is installed. Refer to README.md for more
details. [Default is /usr/local/cuda]:
Please specify the cuDNN version you want to use. [Leave empty to use system default]: 5
Please specify the location where cuDNN 5 library is installed. Refer to README.md for more
details. [Default is /usr/local/cuda]:
Please specify a list of comma-separated Cuda compute capabilities you want to build with.
You can find the compute capability of your device at: https://developer.nvidia.com/cuda-gpus.
Please note that each additional compute capability significantly increases your build time
and binary size.
[Default is: "3.5,5.2"]: 3.0
Setting up Cuda include
Setting up Cuda lib
Setting up Cuda bin
Setting up Cuda nvvm
Setting up CUPTI include
Setting up CUPTI lib64
Configuration finished
```

当用户选择编译生成 GPU 版的 TensorFlow whl 包时，configure 脚本将会为 CUDA 库文件创建软链接。因此，如果用户改变了操作系统的 CUDA 库文件路径，就必须重新运行 configure 脚本配置 TensorFlow 编译选项，以确保使用 Bazel 编译源代码时软链接可用。

(6) 编译 TensorFlow whl 软件包。

首先，我们需要编译 TensorFlow 源代码，生成对应的二进制文件。命令如下：

```
# CPU 版
$ bazel build --config=opt //tensorflow/tools/pip_package:build_pip_package
# GPU 版
$ bazel build --config=opt --config=cuda //tensorflow/tools/pip_package:build_pip_package
```

编译 TensorFlow 源代码需要使用大量内存。如果用户没有特别充足的内存资源，那么可以手动设置内存限制，如 `--local_resources 2048,.5,1.0`。TensorFlow 官方发布的 whl 包均使用 GCC 4 编译。如果用户使用 GCC 5 或更高版本编译，为了兼容老版本的 ABI，建议手动指定参数 `--cxxopt="-D_GLIBCXX_USE_CXX11_ABI=0"`。以上两个手动设置项均是 `bazel build` 命令的输入参数。

接下来，构建 TensorFlow 的 whl 包。`bazel build` 命令执行成功后，会生成 build_pip_package 脚本，用于构建 whl 包。下面展示执行该脚本生成 TensorFlow whl 包的方法：

```
# /tmp/tensorflow_pkg 指定了生成的 TensorFlow whl 包的路径，用户可以替换
$ bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
```

(7) 安装 TensorFlow，具体的安装方法与安装 TensorFlow 官方提供的 whl 包类似：

```
$ sudo pip install /tmp/tensorflow_pkg/tensorflow-1.2.0-py2-none-any.whl
```

2.1.7 Hello TensorFlow

使用前面介绍的方法安装 TensorFlow 后，我们需要验证 TensorFlow 是否安装成功。只要拥有编程背景的人，一定都知道 Hello World!。不出俗套，这里我们也使用 TensorFlow 打印 Hello TensorFlow!，来验证安装的正确性：

```
$ python
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
Hello TensorFlow!
```

如果程序成功打印 Hello TensorFlow!，则说明安装成功。如果出现导入 TensorFlow 软件包错误，则说明安装失败。

一般而言，在 Python 解释器中编写程序并不是一种高效的做法。我们推荐普通用户使用一些成熟的 Python IDE 或 Jupyter Notebook 来开发。Jupyter Notebook 是一套 Python 交互式编程环境，它支持同时编辑多个文件，并以 Web 形式提供服务。下面我们简单介绍 Jupyter Notebook 的安装和使用方法。

(1) 安装并启动 Jupyter Notebook。服务默认运行在本地的 8888 端口，用户在浏览器中输入对应的 URL（http://localhost:8888）即可打开服务：

```
$ pip install jupyter
$ jupyter notebook
[I 23:28:44.100 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 23:28:44.100 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice
to skip confirmation).
```

(2) 创建 Python 交互式环境，图 2-1 给出了相应操作菜单和命令的位置。



图 2-1　在 Jupyter Notebook 中创建 Python 交互式环境

(3) 编写 Python 应用程序，图 2-2 给出了打印 Hello TensorFlow! 的示例。

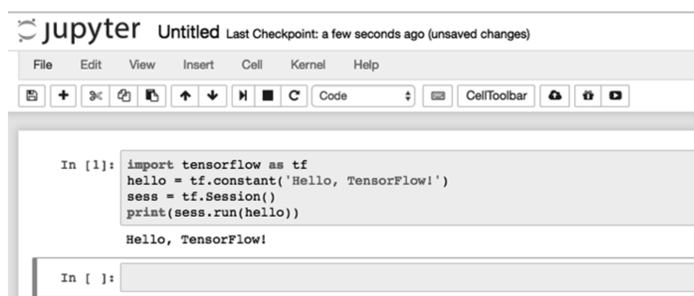


图 2-2　在 Jupyter Notebook 中编写 Python 应用程序

Jupyter Notebook 是所见即所得的交互式编程环境。用户可以指定执行某个输入的代码块，输出结果将直接打印在该代码块下方。

现在，我们已经完成了 TensorFlow 开发环境的搭建。在上述多种安装方法中，用户应该根据自身情况选择合适的方法。当出现安装问题时，可以查看 TensorFlow 官网整理的常见问题和解答（https://www.tensorflow.org/install/install_linux#common_installation_problems）。

2.2　依赖项

TensorFlow 的设计实现引入了很多工具软件和开发库，这些软件和库有的来自 Google 公司，

有的来自第三方开发者；有些以源代码或二进制形式集成到了 TensorFlow 的源代码发布包或二进制安装包，有些则作为编译时或运行时的外部依赖项。审视这些依赖项，可以看出开放的生态系统对于 TensorFlow 发展的重要性。本节挑选几个相对重要的、具有代表性的外部依赖项加以介绍。

2.2.1　Bazel软件构建工具

Bazel（http://bazel.build）是 Google 开源的一套软件构建工具。多数开发人员可能比较熟悉 GNU Autotools、CMake 和 Apache Ant 等经典的构建工具。Bazel 的功能定位与这些软件类似，但具有一些独特的优势，具体如下。

- **多语言支持**：为 C++、Java、Python 等语言提供原生的构建规则，并允许以简易的方式开发针对新语言的扩展。
- **高级构建语言**：构建规则描述语言具有语法简洁、高度抽象，却不失功能丰富的特点，有助于用户快速写出正确的构建规则。
- **多平台支持**：除支持类 Unix 操作系统外，也支持 Android 等移动平台，并在新版本中提供对 Windows 平台的支持。
- **可重现性**：以严格的依赖关系为依据，在提供并行编译和增量编译能力的同时，保证每次编译的结果相同。
- **可伸缩性**：具有高效的增量代码分析能力，支持数十万行代码量级的大规模应用程序构建。

这些优势恰恰满足了 TensorFlow 的技术特征与构建需求，因此 Bazel 成为 TensorFlow 默认的软件构建工具。

Bazel 使用工作空间（workspace）、包（package）和目标（target）三层抽象组织待构建的对象。

- **工作空间**：代表待构建的软件整体，它映射到文件系统中的一个目录，一般是软件源代码包的顶层目录。工作空间使用名为 WORKSPACE 的文件描述元信息，主要的元信息包括工作空间名称与软件的外部依赖项。Bazel 支持在构建时通过本地复制或网络下载等多种方式获取依赖项。
- **包**：代表一组具有逻辑关联的待构建实体，通常映射到软件内部的特定功能模块，并使用独立的目录组织该模块的所有源文件。包具有名为 BUILD 的描述文件，该文件除了描述包的元信息外，其主体内容是一系列目标构建规则的集合。包可以具有层次嵌套关系，典型的嵌套用法是将单元测试包嵌入到对应的功能模块包中。
- **目标**：指包内部管理的、具有独立构建规则的待构建实体，它通常映射到一条或一组编译器命令，用于生成软件模块中独立的目标代码文件（如 .o 文件）或可执行文件。目标的完整名称可以用标签（label）描述，其格式一般为`//[包名]:[目标名]`。例如，TensorFlow 核心层数据流图操作模块的完整目标名（标签）为`//tensorflow/core:ops`。在定义依赖项等场合指定同一包中的目标时，`//[包名]`可以省略。

Bazel 目标构建规则的描述语法比较直观，它由一个规则名称和一组属性构成。规则名称取决于待构建实体所使用的编程语言及期待的输出文件类型。TensorFlow 常用的规则包括用于构建 C++ 库文件的 `cc_library`、用于构建 Python 库文件的 `py_library`，以及用于编译 CUDA 库文件的 `tf_cuda_library` 自定义规则等。规则的属性一般包括构建过程所需的源文件名、依赖项名、编译器选项等。下面给出一组构建规则描述语言的示例：

```
cc_library(
    name = "gradients",
    srcs = ["framework/gradients.cc"],
    hdrs = ["framework/gradients.h"],
    deps = [
        ":cc_ops",
        ":grad_op_registry",
        ":ops",
        ":scope",
        "//tensorflow/core:core_cpu",
        "//tensorflow/core:framework",
        "//tensorflow/core:lib",
        "//tensorflow/core:lib_internal",
    ],
)
```

该规则用于构建 TensorFlow C++ API 中的梯度计算接口，以便生成对应的目标文件。`name`、`srcs`、`hdrs`、`deps` 属性分别定义了目标名称，以及构建过程所需的源文件、头文件和依赖项集合。其中，`cc_ops` 等依赖项来自当前包（tensorflow/cc），`core_cpu` 等依赖项来自核心层包（tensorflow/core）。开发者无须关心编译器的命令行参数，也无须考虑操作系统的目标文件扩展名。Bazel 会自动处理这些细节，生成正确的编译器命令，进而编译出目标文件。Bazel 甚至能够分析 C++ 源代码，检查其实际依赖项与声明依赖项的差异，反馈给开发者，从而避免潜在错误的发生。

使用 Bazel 构建已经定义好目标规则的软件时，只需要运行 `bazel` 命令，该命令的一般参数格式为 `bazel [子命令] [选项] [目标名（标签）]`。常用的子命令包括 `build`（构建）、`test`（测试）和 `clean`（清理）等，常用的选项包括编译器优化开关和详细日志开关等。我们已在 2.1.6 节中看到了使用 Bazel 构建 TensorFlow 安装包的命令。除此之外，在对 TensorFlow 进行二次开发时，开发者常常需要使用 Bazel 构建并运行测试用例，这时 `bazel test` 命令将有用武之地。

2.2.2　Protocol Buffers 数据结构序列化工具

Protocol Buffers（https://developers.google.com/protocol-buffers/）是 Google 开发的一套数据结构序列化工具。自 2008 年开源以来，该工具已被互联网、大数据等领域的大量软件广泛使用。Protocol Buffers 提供一套与操作系统和编程语言无关的数据结构定义语法及序列化表示方法。用户基于 Protocol Buffers 语法编写的数据结构定义文件（.proto 文件）可以通过 Protocol Buffers 编译器（如 protoc）生成 C++、Java、Python 等多种语言的数据结构定义代码。这些代码不但包含基本数据的成员定义，也包含用于访问成员、复制对象，以及对数据结构进行序列化和反序列化操作的辅助方法或类型。将这些自动生成的代码集成到应用程序中，可以实现跨平台、跨语言的

结构化数据交换，同时减轻开发者处理语言差异和编写重复逻辑的开销，有助于提升软件开发时和运行时两方面的效率。

这里我们以 TensorFlow 核心数据结构之一的 TensorProto 为例，说明 Protocol Buffers 的基本用法。TensorProto 是 TensorFlow 内部对张量数据的一种抽象，在 C++、Python API 层及 C++ 核心层均有使用，多用于张量的进程间通信和持久化存储等场合。因此，它对于序列化功能和多语言支持具有明确的需求，很适合使用 Protocol Buffers 定义。TensorProto 的数据结构定义文件是 tensorflow/core/framework/tensor.proto，其关键内容摘录如下：

```
syntax = "proto3";

package tensorflow;

import "tensorflow/core/framework/tensor_shape.proto";
import "tensorflow/core/framework/types.proto";
...

message TensorProto {
    DataType dtype = 1;
    TensorShapeProto tensor_shape = 2;
    int32 version_number = 3;
    bytes tensor_content = 4;
    repeated float float_val = 5 [packed = true];
    ...
};
```

可以看出，.proto 文件主要由语法声明、包名定义、文件引用和消息（即数据结构）定义等部分构成，其中核心内容是消息定义。消息内部可以包含 int32、bytes 等基本类型字段，也可以包含 DataType、TensorShapeProto 等自定义类型字段。repeated 关键字修饰的字段称为重复型字段，类似于编程语言中的数组或向量。所有抽象数据类型均会被映射到目标语言中的实际数据类型。当构建 TensorFlow 源代码包时，Bazel 会调用 Protocol Buffers 编译器，创建名为 tensor.pb.h 和 tensor_pb2.py 的 C++ 与 Python 数据结构定义文件。尽管这些自动生成的文件在实现上相对冗长且晦涩，但其功能和效率一般不亚于普通开发者自行编写的数据结构代码，使用也非常简单。以 C++ 语言为例，protoc 能够生成 TensorProto 类，为之添加构造、析构函数和赋值操作符等辅助函数。对于其中的基本类型字段，用户可以使用与字段同名的方法读取其值，使用 set_[字段名]方法修改其值。基类 Message 提供的 SerializeToString、ParseFromString 方法则可以用于 TensorProto 对象的序列化和反序列化。

为了实现跨平台、跨语言的远程过程调用（RPC），同时简化分布式系统组件间协同机制的设计，Google 在 Protocol Buffers 基础上开发并开源了 gRPC 通信库（http://www.grpc.io）。gRPC 使用 Protocol Buffers 语法描述 RPC 的请求、响应消息类型和函数原型，Protocol Buffers 能够为之生成多种编程语言的客户端与服务端的接口代码——这得益于 Protocol Buffers 的插件机制，该机制可以扩展.proto 文件的语法和代码生成器的功能。gRPC 能够将函数调用方与实现方在空间上解耦，允许双方以异步的时序协同工作。gRPC 底层使用 HTTP/2 作为通信协议，其内部实现了

支持非阻塞异步调用的并发调度引擎。TensorFlow 的进程间通信机制，以及 Serving 等周边组件的交互机制均基于 gRPC 框架开发。下面的示例代码给出了 TensorFlow 的进程间通信框架中，有关数据流图注册操作的 gRPC 服务定义：

2

```
...
message RegisterGraphRequest {
    string session_handle = 1;
    GraphDef graph_def = 2;
    bool has_control_flow = 3 [deprecated = true];
    GraphOptions graph_options = 4;
    DebugOptions debug_options = 5;
}

message RegisterGraphResponse {
    string graph_handle = 1;
}
...

rpc RegisterGraph(RegisterGraphRequest) returns (RegisterGraphResponse);
...
```

这段代码来自 tensorflow/core/protobuf 目录下的 worker.proto 和 worker_service.proto 文件。Protocol Buffers 及其 gRPC 插件以这些文件作为输入，为 TensorFlow 生成 worker 服务的 C++ 接口代码。

Protocol Buffers 和 gRPC 均是 TensorFlow 编译时集成的依赖项。用户在使用 TensorFlow 二进制包时，无须独立安装这两个软件。Bazel 在构建 TensorFlow 源代码包时，能够自动下载并编译这两个软件。

2.2.3 Eigen线性代数计算库

Eigen（http://eigen.tuxfamily.org）是一套基于 C++ 模板技术开发的线性代数计算库。该库由众多开源贡献者共同维护，托管于 TuxFamily 社区，现已被包括 TensorFlow 在内的大量科学和工程计算类软件使用。作为一款基础的数学库，Eigen 具有通用性、高效性、易用性和可靠性等特征。它支持复数、向量、矩阵等多种数据类型，提供大量经典矩阵算法和几何算法的实现，在算法实现中充分利用了主流 CPU 指令集和编译器的高级优化特性。Eigen 仅依赖于标准 C++ 库，这使得它能够兼容多种编译器和操作系统。基于 C++ 模板的技术方案一方面可以为数据类型的扩展和算法的特化实现提供可能，有助于增强库的灵活性；另一方面也可以通过元编程（metaprogramming）技巧将部分计算量由运行时迁移到编译时，有助于提升软件效率。尽管 Eigen 的功能是自包含的，然而它同样支持使用其他计算加速技术提升自身的性能。例如，可以使用 OpenMP 实现多线程并行计算，也可以使用 Intel MKL 实现算法向量化加速。

Eigen 社区对于开源贡献者非常开放。项目源代码包的 unsupported 目录包含了一系列第三方贡献的、尚未由官方提供支持的组件。这些第三方组件既包括对数据结构和算法的扩展，也包括服务于计算逻辑的功能支撑组件，例如用于实现线程调度的线程池组件以及用于接入加速器硬件的适配层组件等。

TensorFlow 使用了 Eigen 提供的多种组件，这些组件大多用于在 CPU 和 OpenCL GPU 设备上实现 TensorFlow 的计算类操作。为了在精度要求不高的场景下节约计算和存储开销，TensorFlow 通常使用 16 位的 `Eigen::half` 浮点类型保存参数。针对矩阵存储和计算需求，TensorFlow 引入了 `Eigen::Matrix` 类型以及 `Eigen::PartialPivLU`、`Eigen::HouseholderQR` 等多种分解算法。由于 TensorFlow 以张量作为核心数据结构，它的许多算法实现借助了 Eigen 第三方组件中的 `Eigen::Tensor` 类型及其相关函数。`Eigen::Tensor` 类型提供张量的高效存储和访问能力，以及降维、缩并和卷积等常用算法的实现，这些算法实现已针对多种计算设备分别进行了优化。`Eigen::Tensor` 组件基于 C++ 11 的特性开发，这一点不同于仅依赖 C++ 98 特性的 Eigen 官方组件。

此外，TensorFlow 也使用 Eigen 第三方组件中的线程池模块——`Eigen::ThreadPool`。事实上，这个模块正是来自 Google 公司的贡献。`Eigen::ThreadPool` 能够以线程池抽象有效管理多核 CPU 的计算资源，将计算任务有序地安排在不同线程上并发执行，并在任务量超过资源量的情况下对任务进行排队。为了以较低的开发成本接入 OpenCL GPU 设备，TensorFlow 还引入了 Codeplay 公司贡献的 `Eigen::SyclDevice` 设备抽象及其相关组件。这些组件使得 TensorFlow 能够简单地以模板参数替换方式，将 CPU 上的算法迁移到 OpenCL GPU。

Eigen 也是 TensorFlow 编译时集成的依赖项，不需要用户独立安装。

2.2.4　CUDA统一计算设备架构

CUDA（https://developer.nvidia.com/cuda-zone）是 NVIDIA 公司推出的一种用于并行计算的软硬件架构，发布于 2007 年。该架构以通用计算图形处理器（GPGPU）作为主要的硬件平台，提供一组用于编写和执行通用计算任务的开发库与运行时环境。CUDA 架构能够充分利用原本为图形渲染而设计的众核 GPU，发挥其并行处理、浮点计算和可编程流水线的技术优势，从而为计算密集型和数据密集型的任务提供高效的算力支持。CUDA 架构最初在高性能计算领域崭露头角，随后也被引入了互联网和大数据生态系统。自从深度学习技术在工业界流行之后，CUDA 架构因能够很好地适配神经网络算法的并行加速需求，已成为深度学习领域，特别是模型训练过程首选的计算架构。包括 TensorFlow 在内的绝大多数机器学习、深度学习平台原生地提供了对 CUDA 架构及 NVIDIA GPU 的支持。

当把 CUDA 一词作为软件依赖项提及时，我们指的往往是 CUDA 架构中的软件组件，即 NVIDIA 驱动程序和 CUDA 工具包（CUDA Toolkit）。用户可以在 NVIDIA 官方网站下载集成了 NVIDIA 驱动程序、CUDA 开发库和编译器的安装包。除了基本的 CUDA 开发库和编译器外，CUDA 工具包还包括 cuBLAS、cuFFT、cuSOLVER、cuDNN 等高级算法库，以及 IDE、调试器、可视化分析器等开发工具，其中部分组件需要独立安装。CUDA 工具包中的一部分软件组件是开源的，例如基于 LLVM 的 CUDA 编译器。其余非开源的组件大多以免费软件的形式提供给开发者和最终用户。

在 CUDA 架构中，不同层次的软件组件均为开发者提供编程接口，以适应不同类型软件的

开发需求。NVIDIA 驱动层的开发接口（即 cu 开头的函数，也称为 CUDA Driver API）较为底层，暴露了 GPU 的若干种内部实现抽象。这种接口能够对 GPU 的运行时行为进行细粒度控制，有助于提升程序的运行时效率，但缺点在于开发过程烦琐。一般的 GPU 应用程序不会直接使用这一层接口，然而 TensorFlow 内部的 GPU 计算引擎——StreamExecutor 为了追求性能，选择使用这一层接口实现 GPU 任务调度和内存管理等功能。CUDA 开发库的 API（即 cuda 开头的函数，也称为 CUDA Runtime API）是 CUDA 架构中使用最为广泛的接口，功能涵盖 GPU 设备管理、内存管理、事件管理以及与图形处理相关的逻辑。不过，既然有了 StreamExecutor，TensorFlow 就没有大面积地使用这个层次的接口。cuBLAS、cuDNN 等高级算法库提供面向通用计算（如线性代数）或领域专用计算（如神经网络）需求的高层次接口。在这个层次，GPU 设备的很多技术细节已被屏蔽，开发者可以专注于算法逻辑的设计和实现。TensorFlow 面向 NVIDIA GPU 的计算类操作大多基于 cuBLAS 和 cuDNN 接口实现。

对于 TensorFlow 而言，CUDA 工具包是不受 Bazel 管理的外部依赖项。用户想要使用 NVIDIA GPU 加速深度学习时，无论部署 TensorFlow GPU 版本的二进制包还是编译通用的源代码包，都需要事先安装带有 NVIDIA 驱动程序的 CUDA 工具包及 cuDNN 库。

2.3 源代码结构

安装了 TensorFlow，并对其外部依赖项有了基本认识之后，读者或许会好奇 TensorFlow 的内部组件结构与实现原理。分析 TensorFlow 的源代码结构是理解其功能构成和模块组织的良好途径，也是对 TensorFlow 进行二次开发的必要条件。即使不打算从事二次开发，学习 TensorFlow 的源代码结构，了解其基本组件的层级关系，同样有助于建立对软件架构的整体印象，从而为编写正确而高效的算法模型奠定基础。此外，源代码包中提供的某些第三方组件、辅助工具、示例和测试代码在二进制安装包中并不存在。如果需要使用它们，就必须从源代码包中提取。有兴趣深入探索这些“隐藏物件”的读者也应当关注 TensorFlow 的源代码结构。

2.3.1 根目录

TensorFlow 源代码的组织符合 Bazel 构建工具要求的规范。其根目录是一个 Bazel 项目的工作空间，包含了 TensorFlow 的所有源代码、Bazel 构建规则文件，以及一些辅助脚本。根目录下的主要子目录和文件介绍如下（以方括号表示目录名，下同）。

- [tensorflow]：TensorFlow 项目自身的源代码。
- [third_party]：部分第三方源代码以及针对第三方项目的 Bazel 构建规则文件。
- [tools]：Bazel 构建过程所需的环境配置脚本。
- [util]：Bazel 构建过程所需的辅助构建规则文件。
- configure：TensorFlow 源代码包配置脚本，用于在构建源代码包之前设置软件的可选特性。
- WORKSPACE：Bazel 工作空间描述文件，包含项目元信息和部分外部依赖项的下载规则。

除此之外，根目录下还包含一系列纯文本的说明文件，用于向用户介绍 TensorFlow 项目的基本情况。

这里有必要特别说明的是 third_party 目录。尽管 Bazel 会通过外部依赖项下载规则，从网络上获取 TensorFlow 所需的绝大多数第三方源代码，然而很多第三方软件自身不包含 Bazel 构建规则，TensorFlow 需要为它们准备相应的规则文件。这些规则文件不但解决了一些项目中特有的技术问题（例如 OpenCL GPU 相关代码所需的专用工具链），而且解决了集成某些项目涉及的非技术问题（例如需要排除 Eigen 库中的部分 GPL 授权代码）。另外，third_party 目录还包括一些第三方项目的头文件，例如 hdfs.h。这是因为 TensorFlow 构建时只需要这些项目的头文件，不需要相应的源文件或库文件，没有必要下载该依赖项的整个软件包。

2.3.2 tensorflow目录

TensorFlow 项目的源代码主体位于 tensorflow 目录。该目录下的源文件几乎实现了 TensorFlow 的全部功能，同时体现了 TensorFlow 的整体模块布局。它的主要子目录和文件介绍如下。

- [c]：C 语言应用层 API，亦作为 C、C++ 以外的其他语言应用层 API 的实现基础。
- [cc]：C++ 语言应用层 API。
- [compiler]：XLA（Accelerated Linear Algebra）编译优化组件的源代码。
- [contrib]：社区托管的第三方贡献组件（主要组件见 15.1 节）。
- [core]：TensorFlow 核心运行时库的源代码，主要使用 C++ 语言实现。
- [docs_src]：TensorFlow 软件文档（即 TensorFlow 官方网站文档）的 Markdown 源代码。
- [examples]：TensorFlow 应用开发示例代码。
- [g3doc]：旧的文档目录，已弃用。
- [go]：Go 语言应用层 API。
- [java]：Java 语言应用层 API。
- [python]：Python 语言应用层 API。
- [stream_executor]：StreamExecutor 库的源代码，主要使用 C++ 语言实现，用于管理 CUDA GPU 上的计算。
- [tensorboard]：TensorBoard 组件的源代码，主要使用 Python 语言实现，用于深度学习过程可视化。
- [tools]：TensorFlow 构建时和运行时使用的工具程序或脚本。
- [user_ops]：用于存放用户自行开发的数据流图操作，包含一组示例代码。
- BUILD：Bazel 构建规则文件，用于构建 TensorFlow 核心运行时库等组件。
- tensorflow.bzl：Bazel 构建过程所需的辅助脚本，主要用于定义 TensorFlow 特有的构建规则。
- workspace.bzl：Bazel 构建过程所需的辅助脚本，主要用于定义外部依赖项的下载规则。

深度学习算法模型的开发者可以对 Python 等应用层 API 目录进行简单浏览，以便了解 TensorFlow 提供的抽象和接口。TensorFlow 核心层的二次开发者则需要聚焦 core 目录，从而理解其内部设计与实现原理。docs_src 目录提供的软件文档适用于所有用户，它是学习 TensorFlow 基础知识的第一手资料。相比这些常用的目录，tools 目录有时会被用户忽略。该目录包含不少实用工具，涉及搭建 TensorFlow 持续集成和基准测试环境所需的框架代码、在 Docker 容器或 Google 云环境中运行 TensorFlow 所需的脚本、用于编辑和压缩模型文件的数据流图转换器，以及自动升级 Python 应用代码中 TensorFlow API 版本的工具等。

2.3.3　tensorflow/core目录

TensorFlow 核心运行时库的源代码位于 tensorflow/core 目录，其主要子目录介绍如下。

- [common_runtime]：核心库的公共运行时源代码，实现了 TensorFlow 数据流图计算的主要逻辑。
- [debug]：用于核心库调试的组件。
- [distributed_runtime]：核心库的分布式运行时源代码，实现了 TensorFlow 分布式运行模式的主要逻辑。
- [example]：使用 Protocol Buffers 创建自定义数据结构并访问序列化文件的示例代码。
- [framework]：核心库的框架性组件，包含 TensorFlow 编程框架中主要抽象的 C++ 或 Protocol Buffers 定义。
- [graph]：数据流图相关抽象和工具类的源代码。
- [grappler]：Grappler 优化器（一种基于硬件使用成本分析的数据流图优化器）的源代码。
- [kernels]：数据流图操作（Op）针对各类计算设备实现的核函数源代码。
- [lib]：公共基础库，涉及通用数据结构、常用算法的实现，以及多种图形、音频格式的访问接口类。
- [ops]：数据流图操作的接口定义源代码。
- [platform]：用于访问特定操作系统或云服务接口的平台相关代码。
- [protobuf]：数据流图基本抽象以外的序列化数据结构的 Protocol Buffers 源代码，例如 gRPC 接口定义。
- [public]：对应用层可见的公开接口的头文件。
- [user_ops]：用于存放用户自行开发的数据流图操作，包含一组示例代码。
- [util]：核心库内部使用的多种实用工具类或函数的集合，例如用于解析命令行参数和访问环境变量的工具。

随着 TensorFlow 版本的演进，核心运行时的源代码组织也有可能发生变化。这些对最终用户不可见的变化往往不会体现在 TensorFlow 的版本发布说明（release note）中，有兴趣的读者可以自行分析目录结构和文件的变化。

读者可能会注意到，tensorflow 和 tensorflow/core 目录下各有一个 user_ops 目录。官方对二者给出了不同的定位：前者适用于用户以二进制包形式安装 TensorFlow 之后，通过 Bazel 的 `tf_custom_op_library` 规则构建自定义操作的情况；后者适用于用户获取 TensorFlow 源代码包之后，在核心运行时库构建过程中编译自定义操作的情况。

2.3.4 tensorflow/python目录

TensorFlow Python API 的源代码位于 tensorflow/python 目录，其主要子目录和文件介绍如下。

- [client]：TensorFlow 主–从模型中的客户端组件，主要包括会话抽象，用于维护数据流图计算的生命周期。
- [debug]：用于 Python 应用程序调试的组件。
- [estimator]：各类模型评价器（estimator）。
- [feature_column]：特征列（feature column）组件。
- [framework]：Python API 的框架型组件，包含 TensorFlow 编程框架中主要抽象的 Python 语言定义。
- [grappler]：Grappler 优化器的 Python 语言接口。
- [kernel_tests]：数据流图操作的单元测试代码，有助于用户学习各种操作的使用方法。
- [layers]：预置的神经网络模型层（layer）组件。
- [lib]：公共基础库，涉及专用数据结构访问和文件系统 I/O 等。
- [ops]：数据流图操作的 Python 语言接口。
- [platform]：用于访问特定操作系统或云服务接口的平台相关代码。
- [saved_model]：用于访问 TensorFlow 通用模型序列化格式（`SavedModel`）的组件。
- [summary]：用于生成 TensorFlow 事件汇总文件（summary）的组件，以便在 TensorBoard 中可视化计算过程。
- [tools]：若干可独立运行的 Python 脚本工具，涉及访问和优化模型文件等功能。
- [training]：与模型训练过程相关的组件，例如各类优化器（optimizer）、模型保存器（saver）等。
- [user_ops]：用于存放用户自行开发的数据流图操作的 Python 语言接口。
- [util]：Python API 内部使用的多种实用工具类或函数的集合，例如用于处理 Python 2/3 文本兼容性的函数。
- build_defs.bzl：Bazel 构建过程所需的辅助脚本，主要用于定义 Python API 特有的构建规则。
- pywrap_tensorflow.py：间接封装核心库通过 C API 导出的函数，以便在 Python API 内部调用核心库的功能。
- tensorflow.i：对接 C API 的 SWIG 接口描述文件，用于在软件构建时为 Python API 生成核心层的动态链接库。

Python API 的多数模块内部通过 SWIG 工具生成的胶合层代码调用 C API，进而使用 C++ 核心库的功能。另有少量 API 的内部实现代码出于提升执行效率或调用核心库未导出接口的目的，直接使用 C++ 语言开发。Python API 在 TensorFlow 源代码中属于更新相对频繁的部分，读者可以通过 TensorFlow 的版本发布说明追踪 API 的新变化。

2.3.5 安装目录

使用官方二进制包或基于源代码包自行构建出来的二进制包安装 TensorFlow 之时，`pip` 命令会将 TensorFlow 运行时所需的 Python 文件、动态链接库以及必要的依赖项复制到当前 Python 环境的 site-packages 或 dist-packages 目录中。其中，TensorFlow 软件本身的运行时代码会被部署到 tensorflow 子目录。这一目录具有和源代码包的 tensorflow 目录相似的组织结构。对于一般用户而言，二者的主要不同点在于以下几点。

- 安装目录中只包含每个模块的 Python 语言接口文件，不再包含 C++ 源代码。所有使用到的 C++ 源代码已被编译到了 python 子目录下的动态链接库文件中（在 Linux 下为 _pywrap_tensorflow_internal.so）。如果某个模块未提供 Python API，那么相应的子目录不会在安装目录中出现。
- 安装目录中的 python/ops 子目录比同名的源代码子目录增加了一系列名称由 gen_ 开头的 Python 接口文件。这些文件是 TensorFlow 编译脚本自动创建的，它们旨在为 C++ 核心库的一部分数据流图操作提供 Python 编程接口。
- 安装目录比源代码目录多出一个 include 子目录。这个目录包含了 TensorFlow 本身以及 Protocol Buffers、Eigen 等依赖库的 C++ 头文件，允许用户通过编程方式使用核心库的功能。需要注意的是，这些头文件提供的是原 core 目录下的 TensorFlow 核心层 API，而非原 cc 目录下的 C++ 应用层 API。

2.4 小结

本章介绍了 TensorFlow 的发布形式和多种安装方法。我们看到，原生 pip 方式适用于简单的 Python 应用程序开发场景，Anaconda 和 virtualenv 方式适用于需要维护多套 Python 虚拟环境的场合，而源代码编译安装方式适合于对性能有较高要求、需要使用可选特性，或者希望对 TensorFlow 做二次开发的情况。读者可以根据自身的需求，选择合适的 TensorFlow 版本和安装方法，从而构建属于自己的深度学习开发环境。本章亦介绍了 TensorFlow 的 4 个主要外部依赖项——Bazel、Protocol Buffers、Eigen 和 CUDA 的功能与用法，以及 TensorFlow 源代码包的目录结构，这为读者进一步学习 TensorFlow 应用层 API 和核心层原理奠定了基础、理顺了思路。

第 3 章

TensorFlow 基础概念

作为深度学习库中的后起之秀，TensorFlow 与 Theano、Caffe 等前辈软件一样使用基于声明式编程的数据流图作为编程范式。与更为程序员所熟知的结构化编程、面向对象编程相比，这种编程范式具有些许不同的风格。无论初入算法模型开发领域的新人，还是具有其他平台使用经验的老手，都有必要理解并适应 TensorFlow 的编程范式及其概念体系。本章以 TensorFlow Python API 层面的概念为基准，首先对比声明式编程与命令式编程各自的特点，给出数据流图在处理机器学习和深度学习问题上的优势。然后介绍 TensorFlow 数据流图的核心抽象，以及 TensorFlow 的数据载体、模型载体、运行环境和训练工具等关键模块。最后，借助一元线性回归模型的例子，以最佳实践的形式加深读者对 TensorFlow 编程范式的理解。

3.1 编程范式：数据流图

作为一个深度学习库，TensorFlow 采用了更适合描述深度神经网络模型的声明式编程范式，并以数据流图作为核心抽象。相比使用更广泛的命令式编程范式，基于声明式编程的数据流图的好处有代码可读性强、支持引用透明、提供预编译优化能力等，这些都有助于用户定义数学函数或算法模型。本节主要介绍声明式编程与命令式编程各自的特点，以及基于声明式编程的数据流图在深度学习应用上的独特优势。在本节最后，我们通过实例逐步引出 TensorFlow 数据流图的基本概念。

3.1.1 声明式编程与命令式编程

计算机科学的研究者将编程语言或函数库为开发人员提供的程序设计基本风格和典型模式定义为编程范式。声明式编程与命令式编程是两种常见的编程范式，它们的最大区别在于：前者强调“做什么”，后者强调“怎么做”。二者各自具有显著的特点。

- **声明式编程**：结构化、抽象化，用户不必纠结每个步骤的具体实现，而是通过下定义的方式描述期望达到的状态。
- **命令式编程**：过程化、具体化，用户告诉机器怎么做，机器按照用户的指示一步步执行命令，并转换到最终的停止状态。

通常，我们认为声明式编程起源于上世纪中叶的人工智能研究，它包括函数式编程（functional programming，简称 FP）和逻辑式编程（logic programming，简称 LP）等子范式。函数式编程将计算描述为对数学函数的求值，通过 lambda 演算精确表达计算逻辑；逻辑式编程基于一系列事实和规则，通过逻辑推导得出结论。声明式编程比较接近人的思考模式：程序中的变量代表数学中的抽象符号，而不是某一块内存地址；用户将计算过程抽象为函数表达式，将程序的输出定义为函数值。声明式程序按照用户定义的函数对输入数据进行表达式变换和计算，程序最终的输出仅依赖于用户的输入数据。计算过程既不受内部状态影响，也不受外部环境影响。后面介绍声明式编程时会以函数式编程为主。

命令式编程起源于对汇编语言和机器指令的进一步抽象，本身带有明显的硬件结构特征。它通过修改存储器的值、产生副作用的方式实现计算。这里的"副作用"是指一个函数或表达式除了返回值之外，还对外部环境产生附加的影响，例如修改了函数作用域外的变量或输入参数。命令式程序具有内部状态，计算的过程就是状态转换的过程，改变状态的方式就是对存储器中的变量进行赋值操作。

编程是一种输入到输出的转化机制，这两种编程范式提供了截然不同的解决方案。

- **声明式编程**：程序是一个数学模型，输入是自变量，输出是因变量，用户设计和组合一系列函数，通过表达式变换实现计算。
- **命令式编程**：程序是一个有穷自动机，输入是起始状态，输出是结束状态，用户设计一系列指令，通过指令的执行完成状态转换。

在不设任何前提条件时，探讨两种编程范式的优劣是没有意义的。两种编程范式没有高下之分，只有左右之别，它们的特点决定了各自擅长的领域。

- **声明式编程**：擅长基于数理逻辑的应用领域，如深度学习、人工智能、符号计算系统等。
- **命令式编程**：擅长复杂业务逻辑的应用领域，如交互式 UI 程序、操作系统与实用工具软件等。

综上，我们将声明式编程和命令式编程的对比总结成表 3-1。

表 3-1 声明式编程与命令式编程的多角度对比

编程范式	核心思想	实现方法	程序抽象	计算过程	计算单元	变量意义	擅长领域	典型应用
声明式编程	做什么	结构化、抽象化	数学模型	表达式变换	函数	抽象符号	数理逻辑	深度学习
命令式编程	怎么做	过程化、具体化	有穷自动机	状态转换	指令	存储单元	业务逻辑	交互式 UI 程序

3.1.2 声明式编程在深度学习应用上的优势

现有的深度学习系统大多推荐使用 Python 等解释型语言开发应用程序，这类语言普遍对声明式和命令式两种编程范式提供良好的支持。下面我们以解释型语言为例，介绍声明式编程在深

度学习应用上的优势。虽然 C++ 等编译型语言的命令式编程范式也可以通过编译时优化达到其中部分效果，但它们不是深度学习应用开发的主流语言。

对于深度学习系统中以数据流图为主要抽象的编程方式，声明式编程的优势主要包括三点：代码可读性强、支持引用透明和提供预编译优化能力。下面分别展开说明。

1. 代码可读性强

通常，声明式编程范式写出来的代码可读性更强。它以目标而非过程为导向，更接近于数学公式或人类的思维方式。用户想要编写一个较为复杂的算法时，声明式代码一目了然，但命令式代码就会比较烦琐，不容易直观地看出其功能。下面我们以使用 Python 语言计算斐波那契（Fibonacci）数为例，说明两种编程范式的差别。

斐波那契数 F_n 的定义为：$F_1=1$，$F_2=1$，$F_n=F_{n-1}+F_{n-2}$（$n>2$）。使用命令式编程范式求解斐波那契数的典型写法如下：

```
def fib(n):
  a, b = 1, 1
  for i in range(1, n):
    a, b = b, a + b
  return a
```

读者可能需要逐行分析代码，才能够将其同斐波那契数的定义关联起来。

现在将其改写为声明式编程范式的代码，如下所示：

```
fib = lambda x : 1 if x <= 2 else fib(x - 1) + fib(x - 2)
```

我们可以使用 lambda 表达式语法在一行内完成函数定义。这段代码不涉及算法实现细节，更加接近自然语言的表达形式，读者很容易理解它的实际功能。

当然，细心的读者可能会指出，这段声明式代码涉及函数递归调用，性能较差。对于 Python 解释器这种简单的运行时环境，递归调用的性能问题确实存在。但对于专门为函数式编程设计的开发库和运行时环境，一般都会有多种编译优化机制来处理递归展开等问题，这可以自动改善性能。TensorFlow 的数据流图运行时框架也不例外。TensorFlow 推荐使用声明式编程范式创建深度神经网络模型。通过调用丰富的内置抽象，可以让网络易于设计且具有良好的层次感。模型代码的可读性因此得到增强，从而缩短了理解程序的时间。

2. 支持引用透明

函数的副作用会给程序设计带来不必要的麻烦，有可能引入难以查找的错误，并降低程序的可读性。引用透明（referential transparency）的概念与函数的副作用相关，且受其影响。如果一个函数的语义同它出现在程序中的上下文无关，则称它是引用透明的。对于用于表达算法的函数，引用透明的一个推论是函数的调用语句可以被它的返回值取代，而不影响程序语义。

声明式编程没有内部状态，也不依赖于外部环境；输出结果由输入数据唯一确定，与代码上下文无关。用户创建的模型就是一幅数据流图，图的拓扑结构由函数的组合关系定义。每一个函

数都是一个计算单元或模块，对应着图中的一个节点。用户可以选择执行任意的模块组合（子图），以得到不同模型结构的输出结果。同时，用户也可以反复执行同样的模型结构，通过输入不同的数据，得到不同的输出结果，以此实现神经网络训练等迭代计算逻辑。这种设计能够简化基于梯度下降法的深度神经网络模型的训练步骤，因为训练模型就是在相同的模型结构上输入不同的批数据，然后反复计算得到各参数梯度并不断更新模型参数。

可以看出，引用透明的函数更符合数学语言中对函数的定义，而不再像计算机语言中函数概念的原始定义——子程序。因此，引用透明的特性对于实现数学算法是非常友好的。TensorFlow和类似的深度学习库内置了大量的数学函数，如代数计算、数组计算、归约计算、卷积计算、神经网络及图像处理函数等。用户一般不需要从头开始制造轮子，而是通过组合内置的函数，创建自己的算法模型。声明式编程将函数视为与其他数据类型一样的“一等公民”，一个函数本身可以以“闭包”（closure）的形式成为另一个函数的输入，而非像命令式编程那样以函数的输入、输出参数的方式传输状态值。这样一来，程序在正确性保证、运行时优化等方面比命令式实现更具优势。

3. 提供预编译优化能力

对于以数据流图为核心抽象的声明式编程语言或函数库，其运行时环境不像解释型命令式语言那样即刻执行代码，而是类似于编译型命令式语言的语法树生成过程，需要事先编译得到完整的数据流图，然后根据用户选择的子图，输入数据进行计算。因此，声明式编程能够实现多种预编译优化，包括无依赖逻辑并行化、无效逻辑移除、公共逻辑提取、细粒度操作融合等。这里我们以无依赖逻辑并行化为例，说明声明式编程如何提供预编译优化能力。下面的伪代码定义了模型 `E=(A+B)*(C-D)`：

```
A = Variable('A')
B = Variable('B')
C = Variable('C')
D = Variable('D')
E = multiply(add(A, B), subtract(C, D))
f = compile(E)
e = f(A=3, B=2, C=4, D=2)  # e = 10
```

图 3-1 描绘了执行预编译函数 `f = compile(E)`后，程序获取到的计算 `E` 的完整数据流图。

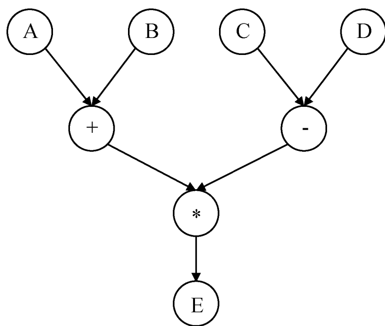


图 3-1　`E=(A+B)*(C-D)`的数据流图示例

如果一个函数的输入值是另一个函数的输出值，则认为前者依赖于后者，对应在数据流图中就是一条后者指向前者的有向边。运行时库获取到整幅数据流图后，能够清楚地理解各个输入、输出之间的依赖关系，故而容易找到可以并行计算的节点。本例中，A+B 和 C-D 之间没有任何依赖关系，因此图上没有对应的有向边，它们可以并行执行。然而，E 必须等待 A+B 和 C-D 计算完成后，才能计算。通过预编译优化技术，TensorFlow 和类似的深度学习库可以在运行时有效提升算法并行度，在多核 CPU 与 GPU 场景下加快程序的运行速度。

3.1.3　TensorFlow数据流图的基本概念

TensorFlow 将数据流图明确地定义为：用节点和有向边描述数学运算的有向无环图。如图 3-2 所示，数据流图中的节点通常代表各类操作（operation），具体包括数学运算、数据填充、结果输出和变量读写等操作，每个节点上的操作都需要分配到具体的物理设备（如 CPU、GPU）上执行。图中的有向边描述了节点间的输入、输出关系，边上流动（flow）着代表高维数据的张量（tensor），这就是 TensorFlow 名称的由来。

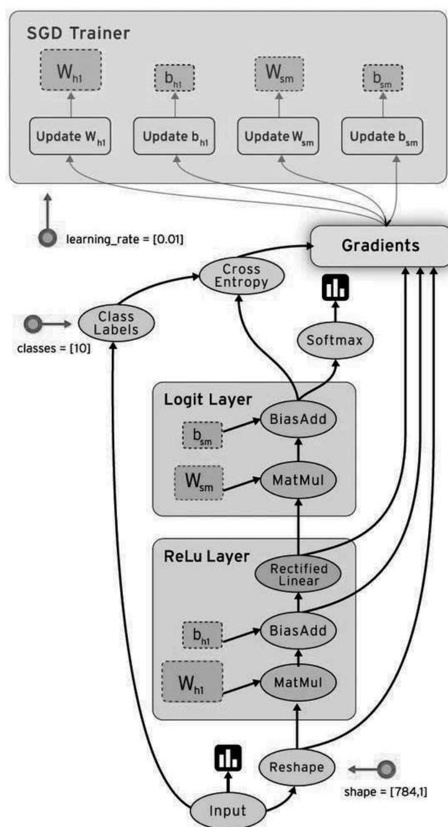


图 3-2　TensorFlow 的数据流图示例（另见彩插）

基于梯度下降法优化求解的机器学习问题，通常都可以分为前向图求值与后向图求梯度两个计算阶段。其中，前向图由用户编写代码完成，主要过程包括定义模型的目标函数（object function）和损失函数（loss function），输入、输出数据的形状（shape）、类型（dtype）等；后向图由 TensorFlow 的优化器（optimizer）自动生成，主要功能是计算模型参数的梯度值，并使用梯度值更新对应的模型参数。

下面分别介绍数据流图中的主要概念，以及数据流图和 TensorFlow 会话的执行原理。

1. 节点

前向图中的节点统一称为操作，它们根据功能可以分为以下 3 类。

- **数学函数或表达式**：比如图 3-2 中的 MatMul、BiasAdd 和 Softmax，绝大多数节点都属于此类。
- **存储模型参数的变量（variable）**：比如图 3-2 中 ReLu Layer 中的 W_{h1} 和 b。
- **占位符（placeholder）**：比如图 3-2 中的 Input 和 Class Labels，它们通常用来描述输入、输出数据的类型和形状等，便于用户利用数据的抽象结构直接定义模型。在数据流图执行时，占位符需要填充对应的数据。

后向图中的节点同样分为以下三类。

- **梯度值**：即经过前向图计算出的模型参数的梯度，比如图 3-2 中的 Gradients。
- **更新模型参数的操作**：比如图 3-2 中的 Update W 和 Update b，它们定义了如何将梯度值更新到对应的模型参数。
- **更新后的模型参数**：比如图 3-2 中 SGD Trainer 内的 W 和 b，与前向图中的模型参数一一对应，但参数值得到了更新，用于模型的下一轮训练。

2. 有向边

数据流图中的有向边用于定义操作之间的关系，它们分为两类：一类用来传输数据，绝大部分流动着张量的边都是此类，在图 3-2 中用实线表示，简称数据边。另一类用来定义控制依赖（control dependency），通过设定节点的前置依赖决定相关节点的执行顺序，在图 3-2 中用虚线表示，简称控制边。

所有的节点都通过数据边和控制边连接。入度为 0 的节点没有前置依赖，可以立即执行；入度非 0 的节点需要等待所有依赖节点执行结束后，方可执行。

3. 执行原理

声明式编程的特点决定了在深度神经网络模型的数据流图上，各个节点的执行顺序并不完全依赖于代码中定义的顺序，而是与节点之间的逻辑关系以及运行时库的实现机制相关。在使用数据边和控制边描述节点依赖关系的基础上，TensorFlow 设计了一套精妙而有序的执行机制来确保数据流图正确执行。这里我们抛开运行时库内部的复杂实现，仅从高层宏观视角审视数据流图的

执行原理。简单来说，数据流图上节点的执行顺序的实现参考了拓扑排序的设计思想。当用户使用 TensorFlow 执行指定数据流图时，其过程可以简述为以下 4 个步骤。

(1) 以节点名称作为关键字、入度作为值，创建一张散列表，并将此数据流图上的所有节点放入散列表中。

(2) 为此数据流图创建一个可执行节点队列，将散列表中入度为 0 的节点加入到该队列，并从散列表中删除这些节点。

(3) 依次执行该队列中的每一个节点，执行成功后将此节点输出指向的节点的入度值减 1，更新散列表中对应节点的入度值。

(4) 重复步骤 (2) 和步骤 (3)，直到可执行节点队列变为空。

下面以图 3-2 所示的数据流图执行过程为例说明。最初可执行节点队列中只有 Input 节点。执行 Input 后，Reshape 和 Class Labels 节点的入度减为 0，加入可执行节点队列。同时，这两个节点将从散列表中移除。接下来，程序将执行 Reshape 和 Class Labels 节点。以此类推，直到可执行节点队列变为空。

TensorFlow 数据流图本身是一个有向无环图，程序结果的正确性依赖于图上节点的执行顺序。通过这套数据流图执行机制，TensorFlow 能够支持复杂、多样化的算法模型。

3.2　数据载体：张量

张量广泛应用于物理学、数学和工程学中。在不同的应用领域，张量具有不同的学术定义。这里援引维基百科的解释：张量是用来表示一些矢量、标量和其他张量之间线性关系的多线性函数，这些线性关系的典型例子有内积、外积、线性映射以及笛卡儿积等。张量的抽象理论是线性代数的分支：多重线性代数。

在 TensorFlow 中，张量是数据流图上的数据载体。为了更方便地定义数学表达式、更准确地描述数学模型，TensorFlow 使用张量统一表示所有数据。在实际计算时，即表达式的转换过程中，模型所对应的表达式中的数据由张量来承载。TensorFlow 提供 `Tensor` 和 `SparseTensor` 两种张量抽象，分别表示稠密数据和稀疏数据。后者旨在减少高维稀疏数据的内存占用。

3.2.1　张量：`Tensor`

在数学中，张量是一种几何实体，广义上可表示任意形式的数据。表 3-2 列出了张量与常见的数据实体的关系，图 3-3 给出了对应的数据实体实例。用于承载数据的张量可以理解为 0 阶标量、1 阶向量和 2 阶矩阵在高维空间上的推广，张量的阶（rank）表示它所描述数据的最大维度。在 NumPy 等数学计算库或 TensorFlow 等深度学习库中，我们通常使用多维数组的形式描述一个张量，数组的维数表示对应张量的阶数。

表 3-2　张量与常见数据实体的关系

阶	数据实体	Python 样例
0	标量	`scalar = 1`
1	向量	`vector = [1, 2, 3]`
2	矩阵（数据表）	`matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
3	数据立方	`tensor = [[[1, 2, 3], [4, 5, 6], [7, 8, 9]], [[10, 11, 12], [13, 14, 15], [16, 17, 18]]]`
n	*n* 阶张量	`...`

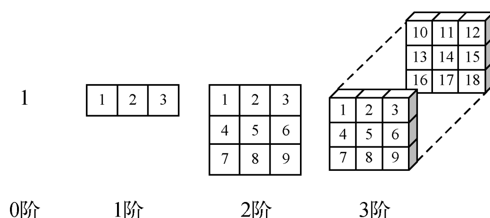


图 3-3　表 3-1 中 Python 样例对应的数据实体示例

张量的阶数决定了其描述的数据所在高维空间的维数。在此基础上，定义每一阶的长度可以唯一确定一个张量的形状。TensorFlow 中的张量形状用列表表示，列表中的每个值依次表示张量各阶的长度。例如，图 3-3 中 3 阶张量各阶的长度分别为 $D_0 = 2$、$D_1 = 3$ 和 $D_2 = 3$，因此它的形状为 [2，3，3]。执行数据流图上的操作时，需要保证同一操作下的张量形状符合计算规则。当用户没有显式设置输出张量的形状时，TensorFlow 内部会根据操作的输入张量进行形状推理（shape inference），以确保操作能够正确执行。

TensorFlow 的张量具有极强的数据表达能力，这既体现在它对高维数据的抽象描述，又体现在它对多样化数据类型的支持。表 3-3 罗列了 TensorFlow 张量支持的数据类型。除了支持常用的浮点数、整数、字符串、布尔型等类型外，也支持复数和量化整数类型。用户可以在 tensorflow/python/framework/dtypes.py 文件中找到相关的代码定义和说明。

表 3-3　TensorFlow 张量支持的数据类型

Dtype 对象	TensorFlow 数据类型	说　明
DT_HALF	tf.float16	半精度浮点数
DT_FLOAT	tf.float32	单精度浮点数
DT_DOUBLE	tf.float64	双精度浮点数
DT_BFLOAT16	tf.bfloat16	裁短浮点数
DT_INT8	tf.int8	8 位有符号整数
DT_INT16	tf.int16	16 位有符号整数
DT_INT32	tf.int32	32 位有符号整数

（续）

Dtype 对象	TensorFlow 数据类型	说　明
DT_INT64	tf.int64	64 位有符号整数
DT_UINT8	tf.uint8	8 位无符号整数
DT_UINT16	tf.uint16	16 位无符号整数
DT_STRING	tf.string	字符串
DT_BOOL	tf.bool	布尔值
DT_COMPLEX64	tf.complex64	单精度复数（实部和虚部均为单精度浮点数）
DT_COMPLEX128	tf.complex128	双精度复数（实部和虚部均为双精度浮点数）
DT_QINT8	tf.qint8	量化的 8 位有符号整数
DT_QINT16	tf.qint16	量化的 16 位有符号整数
DT_QINT32	tf.qint32	量化的 32 位有符号整数
DT_QUINT8	tf.quint8	量化的 8 位无符号整数
DT_QUINT16	tf.quint16	量化的 16 位无符号整数

TensorFlow 的张量在逻辑定义上是数据载体，但在物理实现时是一个句柄，它存储张量的元信息以及指向张量数据的内存缓冲区指针。这样设计是为了实现内存复用。在某些前置操作（生产者）的输出值被输入到多个后置操作（消费者）的情况下，无须重复存储输出值。当一个张量不再被任何操作依赖后，TensorFlow 会释放存储该张量的内存缓冲区。例如，图 3-1 中的 A*B 和 C–D 的计算结果被乘法操作处理之后，存储它们的内存将会被释放。TensorFlow 内部通过引用计数方式判断是否应该释放张量数据的内存缓冲区，这一机制类似于编程语言中的垃圾回收机制。

1. 创建

在 TensorFlow Python API 中，稠密张量抽象是 Tensor 类，它定义在 tensorflow/python/framework/ops.py 文件里。表 3-4 列出了 Tensor 构造方法的完整输入参数，这些参数同时也是张量的属性。

表 3-4　TensorFlow 张量的属性

属性名称	功能说明
dtype	张量传输数据的类型
name	张量在数据流图中的名称
graph	张量所属的数据流图
op	生成该张量的前置操作
shape	张量传输数据的形状
value_index	张量在该前置操作所有输出值中的索引

不过，在一般情况下，用户不需要使用 Tensor 类的构造方法直接创建张量，而是通过操作间接创建张量。典型的张量创建操作包括常量定义操作和代数计算操作。下面的代码给出了使用 constant 和 add 操作创建张量 a、b 和 c 的示例：

```
import tensorflow as tf

a = tf.constant(1.0)
b = tf.constant(2.0)
c = tf.add(a, b)
print([a, b, c])
'''
输出：
[<tf.Tensor 'Const:0' shape=() dtype=float32>,
 <tf.Tensor 'Const_1:0' shape=() dtype=float32>,
 <tf.Tensor 'Add:0' shape=() dtype=float32>]
'''
```

2. 求解

数据流图中的操作输出值由张量承载。如果用户想要求解特定张量的值，则需要创建会话，然后执行张量的 `eval` 方法或会话的 `run` 方法。下面的代码给出了获取张量 `a`、`b`、`c` 值的示例：

```
with tf.Session() as sess:
  print(c.eval())
  print(sess.run([a, b, c]))
'''
输出：
3.0
[1.0, 2.0, 3.0]
'''
```

3. 成员方法

TensorFlow 的 `Tensor` 抽象除了支持多样化的数据类型外，也提供一些成员方法来动态改变张量形状，以及查看张量的后置操作。表 3-5 给出了 TensorFlow 张量的公共成员方法。

表 3-5　TensorFlow 张量的公共成员方法

方法名称	功能说明
`eval`	取出张量值
`get_shape`	获取张量的形状
`set_shape`	修改张量的形状
`consumers`	获取张量的后置操作

4. 操作

TensorFlow 为张量提供了大量操作，以便构建数据流图，实现算法模型。典型的操作如表 3-6 所示。下一节将详细介绍其中的重要操作。

表 3-6　TensorFlow 针对张量提供的典型操作

操作类型	典型操作
一元代数操作	`abs`、`neg` 和 `invert`
二元代数操作	`add`、`multiply` 和 `sub`

（续）

操作类型	典型操作
形状操作	chip、reshape、slice 和 shuffle
归约操作	reduce_mean 和 reduce_sum
神经网络操作	conv、pool、softmax 和 relu
条件操作	cond

5. 典型用例

下面通过一组典型的用例展示张量的创建、求解，以及其他成员方法的使用。示例代码和运行输出如下所示：

```
import tensorflow as tf

a = tf.constant([1, 1])
b = tf.constant([2, 2])
c = tf.add(a, b)
with tf.Session() as sess:
  print("a[0]=%s, a[1]=%s" % (a[0].eval(), a[1].eval()))
  print("c.name=%s" % c.name)
  print("c.value=%s" % c.eval())
  print("c.shape=%s" % c.shape)
  print("a.consumers=%s" % a.consumers())
  print("b.consumers=%s" % b.consumers())
  print("[c.op]:\n%s" % c.op)
'''
输出：
a[0]=1, a[1]=1    # 可以通过下标获取张量的特定部分
c.name=Add:0
c.value=[3 3]
c.shape=(2,)
# 张量 a 和 b 的后置操作均为 add
a.consumers=[<tf.Operation 'Add' type=Add>]
b.consumers=[<tf.Operation 'Add' type=Add>]
# add 操作（即用于生成张量 c 的操作）的属性如下
[c.op]:
name: "Add"
op: "Add"
input: "Const"    # 一个输入值为生成张量 a 的 const 操作
input: "Const_1"  # 另一个输入值为生成张量 b 的 const_1 操作
attr {
  key: "T"
  value {
    type: DT_INT32
  }
}
'''
```

3.2.2　稀疏张量：SparseTensor

TensorFlow 提供了专门用于处理高维稀疏数据的 SparseTensor 类。该类以键值对的形式表

示高维稀疏数据，它包含 indices、values 和 dense_shape 这 3 个属性。其中，indices 是一个形状为 [N, ndims] 的 Tensor 实例，N 表示非零元素的个数，ndims 表示张量的阶数。例如，当 indices=[[0, 2], [1, 3]]（N=2, ndims=2）时，表示 2 阶稀疏张量中索引为 [0, 2] 和 [1, 3] 的元素非零。values 是一个形状为 [N] 的 Tensor 对象，用于保存 indices 中指定的非零元素。dense_shape 是一个形状为 [ndims] 的 Tensor 实例，表示该稀疏张量对应稠密张量的形状。

1. 创建

在 TensorFlow 中创建稀疏张量时，一般可以直接使用 SparseTensor 类的构造方法。示例代码如下：

```
import tensorflow as tf

sp = tf.SparseTensor(indices=[[0, 2], [1, 3]], values=[1, 2], dense_shape=[3, 4])
'''
稀疏张量 sp 的键值对形式为：
[0, 2]: 1
[1, 3]: 2

等价于形状为[3, 4]的 2 阶稠密张量：
  [[0, 0, 1, 0]
   [0, 0, 0, 2]
   [0, 0, 0, 0]]
'''

with tf.Session() as sess:
  # 可以看出，SparseTensor 实例本身由 3 个 Tensor 实例组成
  print(sp.eval())
'''
输出：
SparseTensorValue(indices=array([[0, 2], [1, 3]]),
                  values=array([1, 2], dtype=int32),
                  dense_shape=array([3, 4]))
'''
```

2. 操作

TensorFlow 为稀疏张量提供了一些专门的操作，这样用户能够像处理稠密张量那样处理稀疏张量。典型的操作如表 3-7 所示。

表 3-7 TensorFlow 针对稀疏张量提供的典型操作

操作类型	典型操作
转换操作	sparse_to_dense、sparse_to_indicator 和 sparse_merge
代数操作	sparse_add、sparse_softmax、sparse_tensor_dense_matmul 和 sparse_maximum
几何操作	sparse_concat、sparse_reorder、sparse_split 和 sparse_transpose
归约操作	sparse_reduce_sum 和 sparse_reduce_sum_sparse

3. 典型用例

下面我们通过一组典型的用例展示稀疏张量的创建，以及归约操作的调用方法。示例代码和运行输出如下所示：

```
import tensorflow as tf

x = tf.SparseTensor(indices=[[0,0], [0,2], [1,1]], values=[1,1,1], dense_shape=[2,3])
# 稀疏张量对应的稠密张量为[[1, 0, 1], [0, 1, 0]]
reduce_x = [tf.sparse_reduce_sum(x),                          # => 3
            tf.sparse_reduce_sum(x, axis=1),                  # => [2, 1]
            tf.sparse_reduce_sum(x, axis=1, keep_dims=True), # => [[2], [1]]
            tf.sparse_reduce_sum(x, axis=[0, 1])]             # => 3
with tf.Session() as sess:
  print(sess.run(reduce_x))
'''
输出：
[3,
 array([2, 1], dtype=int32),
 array([[2], [1]], dtype=int32),
 3]
'''
```

3.3　模型载体：操作

TensorFlow 的算法模型由数据流图表示，数据流图由节点和有向边组成，每个节点均对应一个具体的操作。因此，操作是模型功能的实际载体。数据流图中的节点按照功能不同可以分为以下 3 种。

- **计算节点**：对应的是无状态的计算或控制操作，主要负责算法逻辑表达或流程控制。
- **存储节点**：对应的是有状态的变量操作，通常用来存储模型参数。
- **数据节点**：对应的是特殊的占位符操作，用于描述待输入数据的属性。

本节中，我们将依次介绍 TensorFlow 中 3 种节点及其对应操作的定义、功能和典型使用方法。

3.3.1　计算节点：Operation

计算节点对应的计算操作抽象是 `Operation` 类。计算节点的入边代表输入张量，出边代表输出张量。每个节点对输入张量进行特定的数学运算或流程控制，然后将结果输出到后置的节点。`Operation` 类定义在 tensorflow/python/framework/ops.py 文件中，它提供获取操作的名称、类型、输入张量、输出张量等基本属性的方法。表 3-8 列出了计算操作的主要属性。

表 3-8　计算操作的主要属性

属性名称	功能说明
name	操作在数据流图中的名称
type	操作的类型名称

（续）

属性名称	功能说明
inputs	输入张量列表
control_inputs	输入控制依赖列表
outputs	输出张量列表
device	操作执行时使用的设备
graph	操作所属的数据流图
traceback	操作实例化时的调用栈

用户在编写数据流图的算法逻辑时，通常不需要显式构造 Operation 实例，只需要使用 TensorFlow 提供的各种操作函数来定义计算节点。这些函数执行后，TensorFlow 内部会自动构造相应的 Operation 实例。这些操作函数一般定义在 tensorflow/python/ops 目录下的各个文件。表 3-9 分类整理了 TensorFlow Python API 提供的典型操作。

表 3-9　TensorFlow Python API 提供的典型操作

操作类型	典型操作
基础算术	add、multiply、mod、sqrt、sin、trace、fft 和 argmin
数组运算	size、rank、split、reverse、cast、one_hot 和 quantize
梯度裁剪	clip_by_value、clip_by_norm 和 clip_by_global_norm
逻辑控制和调试	identity、logical_and、equal、less、is_finite 和 is_nan
数据流控制	enqueue、dequeue、size、take_grad 和 apply_grad
初始化操作	zeros_initializer、random_normal_initializer 和 orthogonal_initializer
神经网络运算	convolution、pool、bias_add、softmax、dropout 和 erosion2d
随机运算	random_normal、random_shuffle、multinomial 和 random_gamma
字符串运算	string_to_hash_bucket、reduce_join、substr 和 encode_base64
图像处理运算	encode_png、resize_images、rot90、hsv_to_rgb 和 adjust_gamma

用户可以在 Python 解释器中查看操作函数所属的命名空间，进而推断其所在的 Python 文件。方法如下：

```
>>> import tensorflow as tf
>>> help(tf.reshape)
Help on function reshape in module tensorflow.python.ops.gen_array_ops:
...
>>> help(tf.matmul)
Help on function matmul in module tensorflow.python.ops.math_ops:
...
>>> help(tf.add)
Help on function add in module tensorflow.python.ops.gen_math_ops:
...
```

命名空间带有 gen_ 前缀的操作一般由 C++ 核心层实现，并在 TensorFlow 编译时生成对应的

gen_*.py 文件。因此，在 TensorFlow 项目源代码目录中，无法找到这些 Python 文件。感兴趣的读者可以在 TensorFlow 安装之后的 Python site-packages（或 dist-packages）软件包目录下找到对应的文件。

下面我们以 c=a+b 中的 add 操作为例，说明计算操作的执行过程和实现原理。示例代码如下：

```
import tensorflow as tf
# 创建名字作用域 AddExample
with tf.name_scope("AddExample"):
  # 创建变量 a 和 b
  a = tf.Variable(1.0, name='a')
  b = tf.Variable(2.0, name='b')
  # 使用 add 操作函数创建张量 c
  c = tf.add(a, b, name='add')
  print(c)
'''
输出：
Tensor("add:0", shape=(), dtype=float32) # 张量 c 的属性
'''
```

我们能够成功打印输出张量 c 的属性,但是无法直接观察到内部自动构造的 add 操作的属性。事实上，这个操作的一部分属性在代码中已经显式体现出来了：操作名称为 add，操作类型也为 add；输入张量是 a 和 b，它们的值分别为 1.0 和 2.0；输出张量是 c；没有控制依赖输入。为了观察代码中未体现的细节，TensorFlow 为用户提供了查看数据流图的可视化工具——TensorBoard。我们使用 TensorBoard 将 c=a+b 对应的数据流图渲染了出来，如图 3-4 所示。第 6 章将详细介绍 TensorBoard 的使用方法，现在让我们将目光暂时聚焦到 TensorFlow 计算操作。

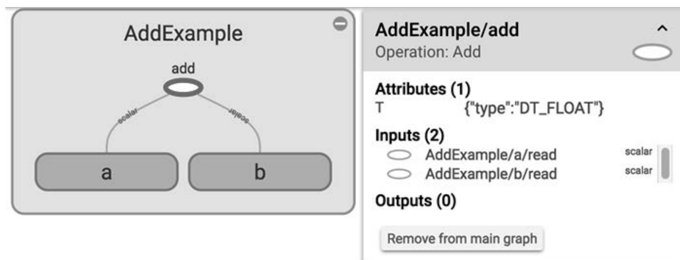


图 3-4　使用 TensorBoard 可视化 c=a+b 的数据流图（左）和 add 操作的属性列表（右）

读者不妨基于图 3-4 思考以下 3 个问题。

(1) 变量 a 和 b 是如何转换为标量（scalar）并传输给 add 操作的？

(2) 张量和标量之间是什么关系？

(3) 在 add 操作属性列表中，Inputs 下的 AddExample/a/read 是什么意思？

在图 3-4 所示的例子中，左图被选中的节点对应的是 add 操作，它的两条输入边上流动的数据的类型均为 scalar。与 add 节点相连的 a 节点保存着变量 a，展开 a 节点，可得到如图 3-5 所

示的变量 `a` 的内部结构（下面简称 a 子图）。可以看出，变量 `a` 其实是由 (a)、Assign、read 和 initial_value 这四个子节点组合而成。当执行 `c = tf.add(a, b, name='add')` 时，`add` 操作通过调用 a 子图中的 read 子节点，将变量 `a` 转换为标量传输给 `add` 操作。上节提到过，标量是一种特殊的张量，即 0 阶张量，所以本质上 add 节点的输入边上流动的仍然是张量表示的数据。看到这里，AddExample/a/read 的含义也就明确了，它表达了一个输入节点的层次结构：AddExample 是名字作用域，`a` 是作用域中的变量，而 read 是 `a` 内部的子节点。

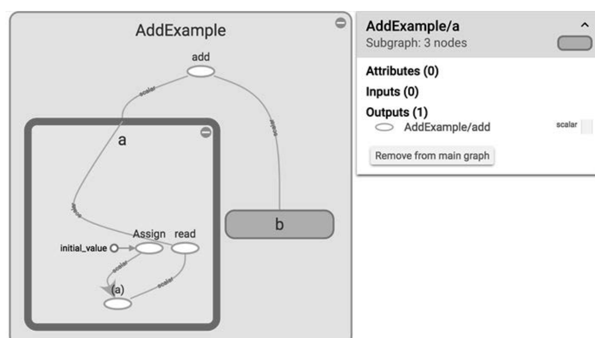


图 3-5　`c=a+b` 的数据流图局部展开（左）和 a 子图的属性列表（右）

在数据流图计算开始之前，用户通常需要执行 `tf.global_variables_initializer` 函数来进行全局变量的初始化。其本质就是将 initial_value 传入 Assign 子节点，实现对变量的初次赋值。图 3-6 展示了本例中 initial_value 子节点的属性列表。

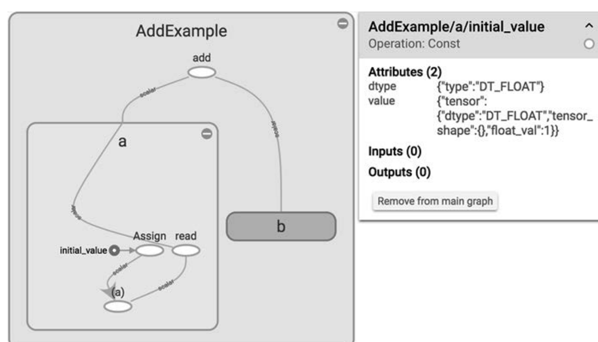


图 3-6　`c=a+b` 的数据流图局部展开（左）和 initial_value 子节点的属性列表（右）

3.3.2　存储节点：Variable

存储节点作为数据流图中的有状态节点，其主要作用是在多次执行相同数据流图时存储特定的参数，如深度学习或机器学习的模型参数。对于无状态节点，其输出由输入张量和节点操作共同确定。对于有状态的节点，如存储节点，其输出还会受到节点内部保存的状态值影响。

1. 变量

TensorFlow 数据流图上的存储节点抽象是 `Variable` 类，我们通常称其为变量。如表 3-10 所示，`Variable` 类提供了变量的名称、数据类型、形状、初始值和初始化操作等属性。感兴趣的读者可以在 tensorflow/python/ops/variables.py 文件中找到 `Variable` 类的定义。

表 3-10　TensorFlow 变量的主要属性

属性名称	功能说明
name	变量在数据流图中的名称
dtype	变量的数据类型
shape	变量的形状
initial_value	变量的初始值
initializer	计算前为变量赋值的初始化操作
device	存储变量的设备
graph	变量所属的数据流图
op	变量操作

如图 3-5 所示，作为存储节点的变量不是一个简单的节点，而是一幅由多个子节点构成的子图。一个变量通常由如下四种子节点构成：

- 变量初始值；
- 更新变量值的操作；
- 读取变量值的操作；
- 变量操作。

在本例的 a 子图中，它们分别对应 inital_value、Assign、read 和 (a) 节点。其中，前三种节点对应的都是无状态操作，而变量操作节点对应的是有状态操作。为了便于区分，TensorBoard 在渲染数据流图时，为有状态操作添加了一对括号。本节的重点是理清变量和变量操作的区别。关于变量的具体使用方法，我们将在 4.2 节中专门介绍。

2. 变量操作

变量操作是 TensorFlow 中的一类有状态操作，用于存储变量的值。变量操作对应的操作函数是 tensorflow/python/ops/state_ops.py 文件中定义的 `variable_op_v2`，其代码如下所示：

```
def variable_op_v2(shape, dtype, name="Variable", container="", shared_name=""):
  """创建变量操作"""
  return gen_state_ops._variable_v2(shape=shape,
                                    dtype=dtype,
                                    name=name,
                                    container=container,
                                    shared_name=shared_name)
```

构造变量操作时，需要给定其存储变量的形状与数据类型。

每个变量对应的变量操作对象在变量初始化时构造。变量支持两种初始化方式。

- **初始值**。用户输入初始值完成初始化。如果没有显式指定初始值，TensorFlow 会根据变量的数据类型进行默认初始化。
- VariableDef。用户使用 Protocol Buffers 定义的变量完成初始化，这通常适用于继续训练时从文件系统中恢复模型参数的场景。

这两种初始化方式分别实现于 Variable 类的私有成员方法 _init_from_args 和 _init_from_proto。用户只能选择其中一种方法进行初始化，否则会导致 ValueError 错误。

图 3-5 中的 a 子图显然属于第一种情况。当我们创建变量 a 时，_init_from_args 方法内部调用了 state_ops.variable_op_v2 函数，创建了该变量对应的变量操作，并将其存储在 Variable 实例的私有成员变量 _variable 中。下面是从 _init_from_args 方法中摘取的相关代码片段：

```
class Variable(object):
  def _init_from_args(...):
    ...
    self._variable = state_ops.variable_op_v2(
        shape,
        self._initial_value.dtype.base_dtype,
        name=name)
```

在本例中，(a) 节点其实就是变量 a 的私有成员变量 _variable，即变量 a 的变量操作。

图 3-7 给出了 TensorBoard 对 (a) 节点属性的解释。可以看到，(a) 节点实际对应 VariableV2 操作，它拥有四个属性：container、dtype、shape 和 shared_name。(a) 节点内部存储变量 a 的值，当用户想要读取或更改 a 的值时，均需要经由 read 或 Assign 节点。

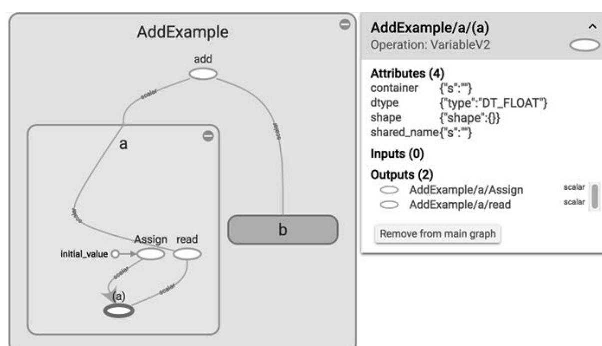


图 3-7 c=a+b 的数据流图局部展开（左）和 (a) 节点的属性列表（右）

3. read 节点

现在我们已经理清了变量和变量操作的区别，明白了变量操作可以用来存储变量的值。下面我们通过解释 read 节点的实现原理，加深读者对于变量、变量操作和变量值的理解。

从 Variable 类的源代码可以看出，读取变量内部存储的值的成员方法是 read_value。该方

法在内部调用了 `identity` 操作，并将操作节点名称设置为 `read`。`identity` 操作接受一个变量作为输入，并返回当前上下文中变量的值。`read_value` 方法的代码如下所示：

```
def read_value(self):
  """Returns the value of this variable, read in the current context."""
    return array_ops.identity(self._variable, name="read")
```

图 3-8 展示了这一过程的内部实现。可以看出，被选中的 read 节点实际对应的是 `identity` 操作。

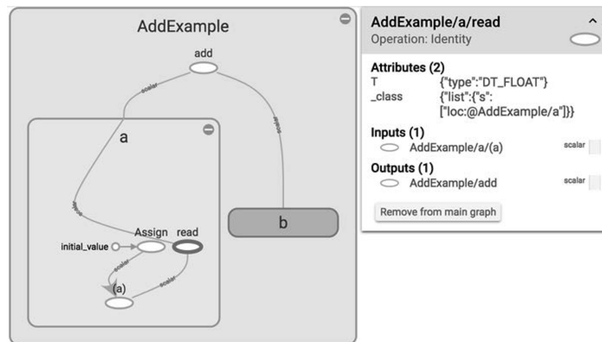


图 3-8　`c=a+b` 的数据流图（左）和 read 节点的属性列表（右）

用户无须显式调用变量的 `read_value` 方法来获取变量值。在数据流图的计算过程中，需要获取这些变量值的后置操作会在内部自动完成相关调用。综上，当用户执行 `c = tf.add(a, b, name='add')` 时，`add` 操作获取变量 `a` 的完整调用栈如图 3-9 所示。这里的 `convert_to_tensor` 方法用于尝试将多种形式的输入转换为张量，因为 TensorFlow 操作的输入不仅支持张量，同时也支持列表、数组和标量等。

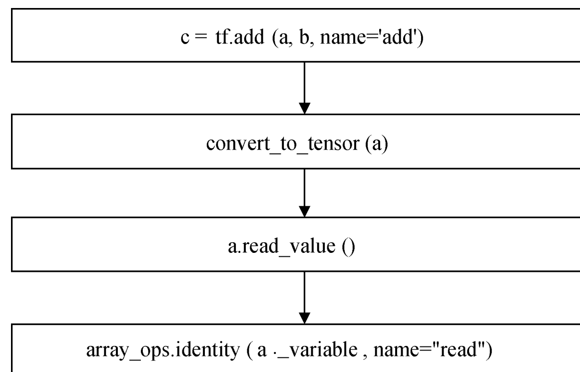


图 3-9　`add` 操作获取变量 `a` 的调用栈

通过上面的分析，我们可以得到如下结论：变量是有状态的节点，其内部的变量操作长期保存变量对应的值。变量值的生命周期与数据流图相同，在数据流图执行过程中始终存在。普通节

点没有内部状态，即不长期保存任何值。它们操作的对象是输入、输出边上流动的张量，这些张量都是临时创建的。当依赖它们的所有操作执行完成后，这些节点及临时张量的内存便会被释放。

3.3.3 数据节点：Placeholder

TensorFlow 数据流图描述了数学模型的计算拓扑，其中各个操作节点都是抽象的函数映射或数学表达式。换句话说，数据流图本身是一个具有计算拓扑和内部结构的“壳”。在用户向数据流图填充数据前，图中并没有真正执行任何计算。

通常，用户在创建模型时已经明确输入数据的类型和形状等属性，而模型的第一步计算很可能就需要使用这些输入数据。TensorFlow 数据节点的作用便是定义待输入数据的属性，使得用户可以描述数据特征，从而完成模型创建。当数据流图执行时，TensorFlow 会向数据节点填充（feed）用户提供的、符合定义的数据。TensorFlow 内部将填充的数据转换为张量后，按照数据流图的拓扑结构执行计算。因此，在创建模型的阶段，用户不需要向数据流图输入任何数据。但在实际计算时，如果没有填充正确的数据，程序就会出错。

TensorFlow 数据节点由占位符操作（placeholder operation）实现，它对应的操作函数是 `tf.placeholder`。针对稀疏数据，TensorFlow 亦提供了稀疏占位符操作（sparse placeholder operation），其操作函数是 `tf.sparse_placeholder`。表 3-11 列出了这两个函数的输入参数。

表 3-11 TensorFlow 占位符和稀疏占位符操作函数的输入参数

属性名称	功能说明
name	占位符操作在数据流图中的名称
dtype	填充数据的类型
shape	填充数据的形状

典型用例

下面我们分别列举占位符和稀疏占位符操作的典型使用示例，以帮助用户快速掌握它们的使用方法和诀窍。

在下面的代码中，x 是由占位操作符创建的 2 阶张量，其形状为 [2, 2]，数据类型为单精度浮点数；y 是 matmul 操作输出的张量，它以 x 作为输入参数。根据依赖控制，用户需要先填充满足 x 定义的张量，才能使得 y 开始执行。然而，这段代码没有为 x 填充数据，所以 TensorFlow 会抛出参数不可用错误（InvalidArgumentError），提示用户为 x 填充符合占位符约束的张量：

```
import tensorflow as tf
import numpy as np

with tf.name_scope("PlaceholderExample"):
  # 定义形状为[2, 2]的单精度浮点数矩阵
  x = tf.placeholder(tf.float32, shape=(2, 2), name="x")
  y = tf.matmul(x, x, name="matmul")
```

```
with tf.Session() as sess:
  # 直接执行matmul操作获取张量y的值时，会报错，因为没有为x填充数据
  print(sess.run(y))
'''输出：
InvalidArgumentError : You must feed a value for placeholder tensor 'PlaceholderExample/x'
with dtype float and shape [2,2]
'''
```

图 3-10 是使用 TensorBoard 可视化上面代码所得数据流图的结果。从选中的 matmul 节点的属性来看，该节点依赖于输入张量 x。如果用户想要正确执行 matmul 节点，那么必须向 x 填充数据。

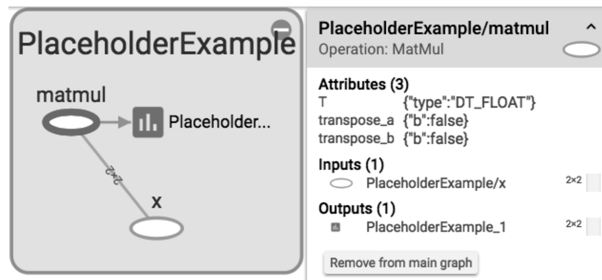


图 3-10　使用 TensorBoard 可视化 y=matmul(x, x) 所得的数据流图（左）和 matmul 节点的属性列表（右）

为了解决输入参数不可用的问题，一种方法是调用 NumPy 库的随机数生成方法 np.random.rand（默认的数据类型为单精度浮点数），并通过 Session.run 方法的 feed_dict 参数将生成的随机数填充到 x。这样一来，程序便可以正确执行，并输出计算结果。需要修改的代码如下所示：

```
with tf.Session() as sess:
  # 将符合定义的数据填充进x后，正确执行
  rand_array = np.random.rand(2, 2)
  print(sess.run(y, feed_dict={x: rand_array}))
'''
输出：
[[ 0.27517948  0.1988095 ]
 [ 0.07253421  0.37904623]]
'''
```

最后，我们给出一段稀疏占位符操作的典型用例：

```
import tensorflow as tf
import numpy as np

# 没有显式指定数据形状，表示可以填充任意形状的单精度浮点数稀疏张量
x = tf.sparse_placeholder(tf.float32)
y = tf.sparse_reduce_sum(x)

with tf.Session() as sess:
  # 设置非零元素的索引为[3, 2, 0]和[4, 5, 1]
  indices = np.array([[3, 2, 0], [4, 5, 1]], dtype=np.int64)
  # 设置索引为[3, 2, 0]和[4, 5, 1]元素的值分别为1.0和2.0
  values = np.array([1.0, 2.0], dtype=np.float32)
```

```
  # 设置稀疏张量对应的稠密张量形状为[7, 9, 2]
  shape = np.array([7, 9, 2], dtype=np.int64)
  # 向 x 填充稀疏张量
  print(sess.run(y, feed_dict={
    x: tf.SparseTensorValue(indices, values, shape)}))
  # 向 x 填充张量 3 元组(indices, values, shape)
  print(sess.run(y, feed_dict={x: (indices, values, shape)}))
  # 向 x 填充 NumPy 多维数组
  sp = tf.SparseTensor(indices=indices, values=values, dense_shape=shape)
  sp_value = sp.eval()
  print(sess.run(y, feed_dict={x: sp_value}))
'''
输出：
3.0
3.0
3.0
'''
```

在该例中，x 是稀疏占位符操作返回的稀疏张量，其元素数据类型为单精度浮点数，形状未定。y 是对 x 中的所有元素进行归约求和计算的结果。这里定义了 3 个 NumPy 数组——indices、values 和 shape，分别表示稀疏张量中非零元素的索引、值，以及对应的稠密张量形状。我们以 3 种不同的格式向 x 填充数据，并执行数据流图，程序均能输出符合预期的结果。

3.4　运行环境：会话

TensorFlow 数据流图描述了计算的拓扑结构和所需的数据属性，但数据流图本身仅是一个“壳”。只有在图中填充了数据、选择了待求解的张量，并执行了相应的计算操作后，才能取得最终结果。TensorFlow 会话为用户提供了上述计算过程的运行环境，它本质上是在维护一段运行时上下文。会话通过提取和切分数据流图、调度并执行操作节点，将抽象的计算拓扑转化为设备上的执行流，从而帮助用户完成计算任务。本节主要介绍 TensorFlow 会话的定义和使用方法，并对比普通会话和交互式会话的异同。

3.4.1　普通会话：Session

TensorFlow 会话提供求解张量和执行操作的运行环境。它是发放计算任务的客户端，所有计算任务都由它分发到其连接的执行引擎完成。在 Python API 中，会话由 tensorflow/python/client/session.py 文件定义的 Session 类实现。一个会话的典型使用流程分为 3 步：(1) 创建会话；(2) 运行会话；(3) 关闭会话。框架性代码如下所示：

```
sess = tf.Session() # (1)创建会话
sess.run(...)       # (2)运行会话
sess.close()        # (3)关闭会话
```

下面我们分别介绍会话的这 3 步典型使用流程。

1. 创建会话

会话为 TensorFlow 数据流图提供运行环境，用户一般通过调用 Session 类的构造方法来创

建会话实例。一个典型的会话实例具有 3 个主要属性：会话连接的执行引擎、加载的数据流图及其启动配置项。如表 3-12 所示，这 3 个属性均是 Session 构造方法的输入参数。

表 3-12　Session 构造方法的输入参数

参数名称	功能说明
target	会话连接的执行引擎
graph	会话加载的数据流图
config	会话启动时的配置项

target 参数的默认值指向进程内引擎（in-process engine），因此用户执行单机任务时无需显式指定它。本节仅分析单机运行的场景。分布式运行模式涉及 TensorFlow 集群和分布式会话的配置，我们将在 5.2 节中专门介绍。graph 参数的默认值指向当前代码中的唯一一幅默认数据流图。当用户没有设置 graph 参数时，会话加载默认数据流图；当用户在代码中定义了多幅数据流图时，则需要通过 graph 显式指定待加载的数据流图。config 参数详细描述了会话的各项配置信息，保证会话启动后能够按照用户希望的状态运行。它由 tensorflow/core/protobuf/config.proto 文件中的 ConfigProto 数据结构定义，用户可使用 ConfigProto 类的构造方法创建会话的配置参数。表 3-13 列出了会话的主要配置参数。

表 3-13　会话的主要配置参数

参数名称	功能说明	数据类型
device_count	设备数量字典，形如<设备类型，最大使用数量>	map<string, int32>
intra_op_parallelism_threads	独立操作并行计算线程数	int32
inter_op_parallelism_threads	阻塞操作并行计算线程数	int32
use_per_session_threads	是否使用一套独立线程池取代全局线程池	bool
session_inter_op_thread_pool	线程池配置参数	ThreadPoolOptionProto
placement_period	重新计算分配节点到设备的周期	int32
device_filters	过滤掉不使用的设备	string
gpu_options	GPU 的配置参数	GPUOptions
allow_soft_placement	在没有 GPU 可用时，是否将操作放置到 CPU 执行	bool
log_device_placement	是否打印设备放置日志	bool
graph_options	数据流图的配置参数	GraphOptions
operation_timeout_in_ms	阻塞操作的最长时限	int64
rpc_options	分布式任务运行时的 RPC 配置参数	RPCOptions

假设用户希望在没有 GPU 时，TensorFlow 能够自动将计算任务转移到 CPU 上运行，并希望通过日志验证放置情况，那么可以使用如下参数创建会话：

```
sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
                                        log_device_placement=True))
```

2. 运行会话

运行会话是指基于数据流图和输入数据，求解张量或执行操作的过程。Session 类用于运行会话的方法是 run，该方法的输入参数见表 3-14。由于会话构造时已经绑定了数据流图，在运行时只需要指定待求解的张量和待填充的数据即可。fetches 参数用于指定用户待获取的张量，即希望经由会话求解的对象；feed_dict 参数用于指定需要填充的数据节点及对应的输入数据。值得一提的是，fetches 参数不仅可以接受张量，而且也能接受操作，因为操作的输出本质上也是张量。

表 3-14 Session.run 方法的输入参数

参数名称	功能说明
fetches	待求解的张量或操作
feed_dict	数据填充字典，形如<数据节点，填充数据>
options	RunOptions 对象，用于设置会话运行时的可选特性开关
run_metadata	RunMetadata 对象，用于收集会话运行时的非张量元信息输出

对于没有数据依赖的张量，调用 Session.run 方法时无需指定数据填充字典。示例代码如下：

```
import tensorflow as tf
# 创建数据流图：c = a * b
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
sess = tf.Session()
# 求解张量 c 的值；没有依赖数据节点，所以无需填充数据
print(sess.run(c))
'''
输出：
30.0
'''
```

对于存在数据依赖的张量，调用 Session.run 方法时，则需要指定数据填充字典。示例代码如下：

```
import tensorflow as tf
# 创建数据流图：z = x * y，其中 x 和 y 均为数据节点
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = x * y
sess = tf.Session()
# 求解张量 z 的值；对数据节点 x 和 y 分别填充输入数据 3.0 和 2.0
print(sess.run(z, feed_dict={x: 3.0, y: 2.0}))
'''
输出：
6.0
'''
```

除了可以使用 Session.run 方法求解张量，我们还在 3.2 节中提到过 Tensor 类的 eval 方法。

事实上,能够实现张量求解的还有 Operation 类的 run 方法。图 3-11 和图 3-12 分别展示了 Tensor.eval 和 Operation.run 方法的调用栈。可以看出,它们的内部实现均调用了 Session.run 方法。

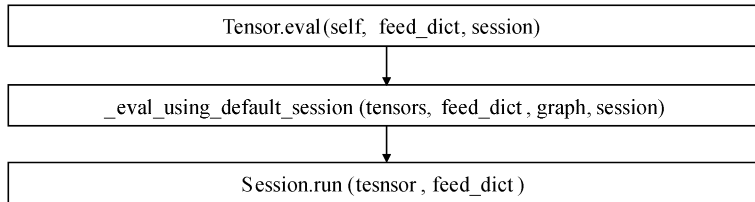


图 3-11 Tensor.eval 方法的调用栈

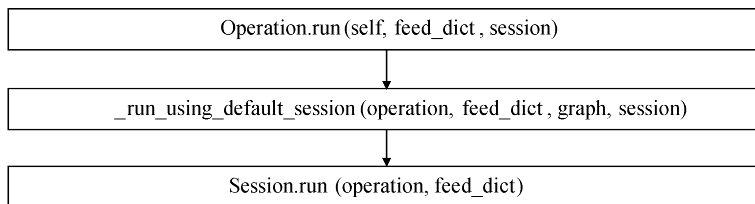


图 3-12 Operation.run 方法的调用栈

相比 Session.run 方法，Tensor.eval 和 Operation.run 方法增加了输入参数 session，它用来指定求解张量和执行操作的会话实例。如果用户没有显式设置 session 参数，TensorFlow 不会自动使用之前创建的会话实例。这时如果直接调用 Tensor.eval 或 Operation.run 方法，程序就会抛出找不到默认会话实例的错误。对于这种场景，用户可以使用 with 语句创建和运行会话。with 语句会隐式调用新建的 Session 对象的 __enter__ 方法，将当前的会话实例注册为默认会话。因此，在 with tf.Session() as sess:语句块中，用户可以使用更简单的参数调用 Tensor.eval 和 Operation.run 方法。示例代码如下：

```
import tensorflow as tf
# 创建数据流图：y = W * x + b，其中 W 和 b 为存储节点，x 为数据节点
x = tf.placeholder(tf.float32)
W = tf.Variable(1.0)
b = tf.Variable(1.0)
y = W * x + b
with tf.Session() as sess:
  tf.global_variables_initializer().run() # Operation.run
  fetch = y.eval(feed_dict={x: 3.0})      # Tensor.eval
  print(fetch)                            # fetch = 1.0 * 3.0 + 1.0
'''
输出：
4.0
'''
```

3. 关闭会话

会话提供了 TensorFlow 数据流图的运行环境，它同时也拥有变量、队列和文件句柄等资源。

当执行完计算任务后，应该主动关闭会话，以便释放这些资源。用户一般可以通过以下两种方式关闭会话。

- 使用 `close` 方法显式关闭会话，例如：

```
sess.close()
```

- 使用 `with` 语句隐式关闭会话，例如：

```
with tf.Session() as sess:
  sess.run(...)
  # Session.__exit__ 方法被隐式调用，进而调用 Session.close 方法关闭会话
```


3.4.2　交互式会话：InteractiveSession

TensorFlow 交互式会话为用户提供类似 shell 的交互式编程环境，它由 `InteractiveSession` 类实现。该类的构造方法不仅完成了交互式会话实例的创建，同时将该实例注册为默认会话，这是交互式会话与普通会话的重要区别。因此，如果用户使用交互式会话作为数据流图的运行环境，便可以不借助 `with` 上下文语句块，直接调用 `Tensor.eval` 和 `Operation.run` 方法求解张量、执行操作。在这种情况下，用户仍需显式调用 `InteractiveSession.close` 方法关闭会话。

使用交互式会话求解张量的示例如下：

```
import tensorflow as tf
# 创建数据流图：c = a * b
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
# 创建交互式会话
sess = tf.InteractiveSession()
# 求解张量 c 的值
print(c.eval())
# 关闭交互式会话
sess.close()
'''
输出：
30.0
'''
```

3.4.3　扩展阅读：会话实现原理

下面简要介绍 TensorFlow Python API 层面会话类的实现原理，帮助读者加深对会话的理解。

`Session` 类与 `InteractiveSession` 类均继承自 `BaseSession` 类，`BaseSession` 类实现了接口类 `SessionInterface` 定义的会话基本属性。这些类的 UML 类图如图 3-13 所示。

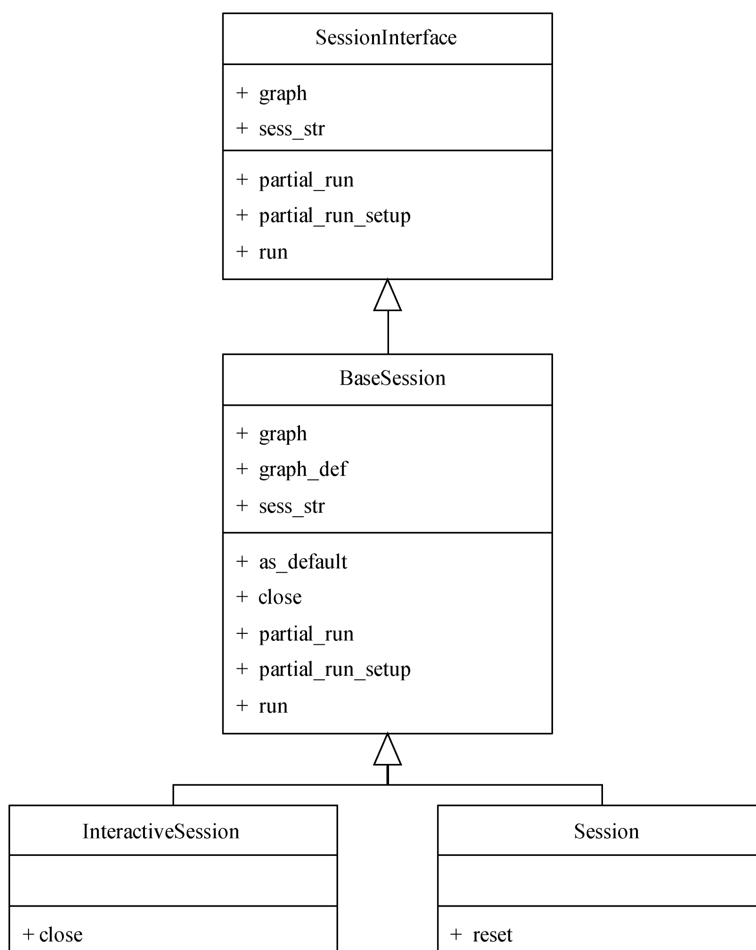


图 3-13　几种会话类的 UML 类图

与 Java 和 C++ 不同，Python 语言没有内置的接口和抽象类，但可以利用异常机制抛出错误，强制子类必须实现特定的属性和成员方法。以下代码展示了 `SessionInterface` 类提供的会话访问接口：

```
class SessionInterface(object):

  # 会话加载的数据流图
  @property
  def graph(self):
    raise NotImplementedError('graph')

  # 会话连接的执行引擎
  @property
  def sess_str(self):
    raise NotImplementedError('sess_str')
```

```
# 求解张量、执行操作
def run(self, fetches, feed_dict=None, options=None, run_metadata=None):
  raise NotImplementedError('run')

# 为 partial_run_setup 方法设置 feeds 和 fetches 参数
def partial_run_setup(self, fetches, feeds=None):
  raise NotImplementedError('partial_run_setup')

# 使用 partial_run 方法设置的 feeds 和 fetches 参数，继续执行计算任务
def partial_run(self, handle, fetches, feed_dict=None):
  raise NotImplementedError('partial_run')
```


BaseSession 类继承了 SessionInterface 类，并新增了 as_default 和 close 方法，这两个方法的作用分别是将当前 BaseSession 实例注册为默认会话，以及关闭当前 BaseSession 实例。同时，还新增了属性 graph_def，它表示序列化后的数据流图，方便用户从文件中加载数据流图继续训练。虽然 BaseSession 类实现了执行数据流图计算任务的方法，但是用户并不能使用这个类，而应使用其子类 Session 和 InteractiveSession。从面向对象的设计思想看，前者实现了两种会话的公共方法，而后两者则实现会话上下文的不同管理方法。

Session 和 InteractiveSession 类分别实现了 reset 和 close 方法。Session 类的 reset 方法主要用于分布式会话的资源释放，包括释放队列和文件句柄资源，以及重置数据流图中的变量。InteractiveSession 类重写了父类的 close 方法，其作用是在关闭交互式会话实例的同时注销加载的数据流图。

3.5 训练工具：优化器

机器学习方法能够在给定的数据中发现潜在的模式，并将发现的模式应用于新数据。机器学习大致分为 3 种类型，分别是监督学习、无监督学习和半监督学习。其中，监督学习是目前工业界应用最广泛的一类方法。典型的监督学习问题由 3 部分组成：模型、损失函数和优化算法。在数据中发现的潜在模式就是人们常说的模型。有的模型可以通过解析式精确定义，如线性回归、逻辑回归、高斯混合模型等；有的模型则不能，如多层感知机、深度神经网络等。

简单来说，模型本身是一系列数学表达式和一组参数的组合。监督学习的模型训练是指在给定数据上不断拟合，以求出一组在测试数据集上使得推理值尽可能接近真实值的模型参数。损失函数是指表达模型推理值同真实值差距的函数，用于评估模型的拟合程度；优化算法则是指使用损失值不断优化模型参数，以尽可能减小损失值的算法。目前，主流的监督学习方法主要采用基于梯度下降的优化算法进行模型训练。本节首先介绍损失函数与优化算法等背景知识，然后讲解 TensorFlow 优化器的定义和使用方法，最后分析 TensorFlow 优化器的实现原理。

3.5.1 损失函数与优化算法

为了更好地理解模型训练和优化器的工作原理，我们首先介绍监督学习中最重要的两个概

念：损失函数和优化算法。

1. 损失函数

损失函数是评估特定模型参数和特定输入时，表达模型输出的推理值与真实值之间不一致程度的函数。损失函数 L 的形式化定义如下：

$$\text{loss} = L(f(x_i;\theta), y_{-i})$$

其中，x_i 是第 i 个输入样本，θ 是模型参数，f 是用于形式化表示模型的数学函数，y_{-i} 是 x_i 对应的真实值。常见的损失函数有平方损失函数、交叉熵损失函数和指数损失函数等，下面分别展示了它们的标准公式：

$$\text{loss} = (y_{-i} - f(x_i;\theta))^2$$
$$\text{loss} = y_{-i} * \log(f(x_i;\theta))$$
$$\text{loss} = \exp(-y_{-i} * f(x_i;\theta))$$

损失函数是一个非负实值函数，值越小说明模型对训练集拟合得越好。使用损失函数对所有训练样本求损失值，再累加求平均可得到模型的经验风险 $R_{\text{emp}}(f)$。换句话说，$f(x)$关于训练集的平均损失就是经验风险，其形式化定义如下：

$$R_{\text{emp}}(f) = \frac{1}{N}\sum_{i=1}^{N} L(f(x_i;\theta), y_{-i})$$

然而，如果过度追求训练数据上的低损失值，就会遇到过拟合问题。训练集通常并不能完全代表真实场景的数据分布。当两者的分布不一致时，如果过分依赖训练集上的数据，面对新数据时就会无所适从，这时模型的泛化能力就会变差。模型训练的目标是不断最小化经验风险。随着训练步数的增加，经验风险将逐渐降低，模型复杂度也将逐渐上升。为了降低过度训练可能造成的过拟合风险，可以引入专门用来度量模型复杂度的正则化项（regularizer）或惩罚项（penalty term）$J(f)$。常用的正则化项有 L0、L1 和 L2 范数。综上，我们将模型最优化的目标替换为健壮性更好的结构风险最小化（structural risk minimization，SRM）。如下所示，它由经验风险项和正则项两部分构成，其中 λ 代表正则项的权重：

$$R_{\text{srm}}(f) = \min \frac{1}{N}\sum_{i=1}^{N} L(f(x_i;\theta), y_{-i}) + \lambda J(\theta)$$

在模型的训练过程中，结构风险不断降低。当 $R_{\text{srm}}(f)$ 小于我们设置的损失值阈值时，则认为此时的模型已经满足需求。因此，模型训练的本质就是在最小化结构风险的同时取得最优的模型参数。最优模型参数的形式化定义如下：

$$\theta^* = \arg\min_\theta R_{\text{srm}}(f) = \arg\min_\theta \frac{1}{N}\sum_{i=1}^{N} L(f(x_i;\theta), y_{-i}) + \lambda\Phi(\theta)$$

2. 优化算法

典型的机器学习和深度学习问题通常都需要转换为最优化问题进行求解。最优化是应用数学

的一个分支，主要研究以下形式的问题：给定一个函数 $f: A \to R$，寻找一个属于集合 A 的元素 x_0，使得对于 A 中的所有 x，$f(x_0) \leqslant f(x)$（最小化），或者 $f(x_0) \geqslant f(x)$（最大化）。这里的函数 f 称为目标函数，A 的元素称为可行解，使得目标函数值最小化的可行解称为最优解。求解最优化问题的算法称为优化算法，它们通常采用迭代方式实现：首先设定一个初始的可行解，然后基于特定的函数反复重新计算可行解，直到找到一个最优解或达到预设的收敛条件。不同的优化算法采用的迭代策略各有不同，有的使用目标函数的一阶导数，如梯度下降法；有的使用目标函数的二阶导数，如牛顿法；有的使用前几轮迭代的信息，如 Adam。基于梯度下降法的迭代策略最简单，它直接沿着梯度负方向，即目标函数减小最快的方向进行直线搜索。这种方法的计算表达式为：

$$x_{k+1} = x_k - \alpha * \mathrm{grad}(x_k)$$

梯度下降法的优点是计算量小，仅计算一阶导数即可；它的缺点是收敛速度慢，只能做到线性收敛。梯度下降法过于简单的策略导致其在优化复杂模型时几乎不会被使用。

下面我们以一个具体的例子来说明损失函数与优化算法的作用。该问题的构成元素如下。

- 模型：

$$y = f(x) = wx+b$$

其中，x 是输入数据，y 是模型输出的推理值，w 和 b 是模型参数。

- 损失函数：

$$\mathrm{loss} = L(y, y_) = L(wx+b, y_)$$

其中，$y_$ 是 x 对应的真实值（标签），loss 为损失函数输出的损失值。

- 优化算法：

$$w \leftarrow w + \alpha * \mathrm{grad}(w) = \alpha \frac{\partial \mathrm{loss}}{\partial w}$$

$$b \leftarrow b + \alpha * \mathrm{grad}(b) = \alpha \frac{\partial \mathrm{loss}}{\partial b}$$

其中，grad(w) 和 grad(b) 分别表示在损失值为 loss 时 w 和 b 对应的梯度值，α 为学习率。

模型的训练过程将上述 3 个部分联系起来。模型 $f(x)$ 的具体形式可以千变万化，但本质上都是输入数据为 x、输出推理值为 y 的数学函数。损失函数的作用是定量描述推理值 y 与真实值 $y_$ 的不一致程度，即求得损失值 loss。利用损失值和模型的拓扑结构，可以计算出模型参数的梯度值 gradients。随后，优化算法以一种高效而合理的方式将梯度值更新到对应的模型参数，完成模型的一步迭代训练。

图 3-14 是使用 TensorBoard 可视化上述模型所得的数据流图的结果。我们有意不展示模型推理值 y 和梯度 gradients 的内部结构细节，目的是希望此图具有泛化的含义，而非代表某个特定的模型。

图 3-14　典型机器学习或深度学习模型的数据流图

3.5.2　优化器概述

优化器是 TensorFlow 实现优化算法的载体，它为用户实现了自动计算模型参数梯度值的功能。TenosrFlow 的优化器根据前向图的计算拓扑和损失值，利用链式求导法则依次求出每个模型参数在给定数据下的梯度值，并将其更新到对应的模型参数以完成一个完整的训练步骤。因此，优化器可以称为 TensorFlow 的训练工具。

为了说明优化算法的实现原理，我们将图 3-14 中 gradients 子图内计算梯度值的结构展开。如图 3-15 所示，gradients 子图是由 TensorFlow 优化器自动生成的梯度计算子图，即后向图。其中 loss 和 y 分别表示对前向图中损失值和推理值求梯度的子图。

图 3-15　计算梯度值的子图内部结构

表 3-15 罗列了 TensorFlow Python API 为用户提供的所有优化器，图 3-16 给出了其中部分典型优化器的类型继承关系。优化器的基类是 `Optimizer`，它定义在 tensorflow/python/training/optimizer.py 文件中。与 `BaseSession` 类的情况类似，用户并不会直接创建 `Optimizer` 类的实例，而是需要创建特定的子类实例。这些优化器均定义在 tensorflow/python/training 目录下。其中，除了同步优化器（Synchronize Replicas Optimizer）是为分布式训练设计的优化器外，其余都是用于单机模型训练的优化器。这些单机优化器经过同步优化器封装之后，也可以应用于分布式训练场景。

表 3-15 TensorFlow 提供的优化器

优化器名称	文件路径
Adadelta	tensorflow/python/training/adadelta.py
Adagrad	tensorflow/python/training/adagrad.py
Adagrad Dual Averaging	tensorflow/python/training/adagrad_da.py
Adam	tensorflow/python/training/adam.py
Ftrl	tensorflow/python/training/ftrl.py
Gradient Descent	tensorflow/python/training/gradient_descent.py
Momentum	tensorflow/python/training/momentum.py
Proximal Adagrad	tensorflow/python/training/proximal_adagrad.py
Proximal Gradient Descent	tensorflow/python/training/proximal_gradient_descent.py
Rmsprop	tensorflow/python/training/rmsprop.py
Synchronize Replicas	tensorflow/python/training/sync_replicas_optimizer.py

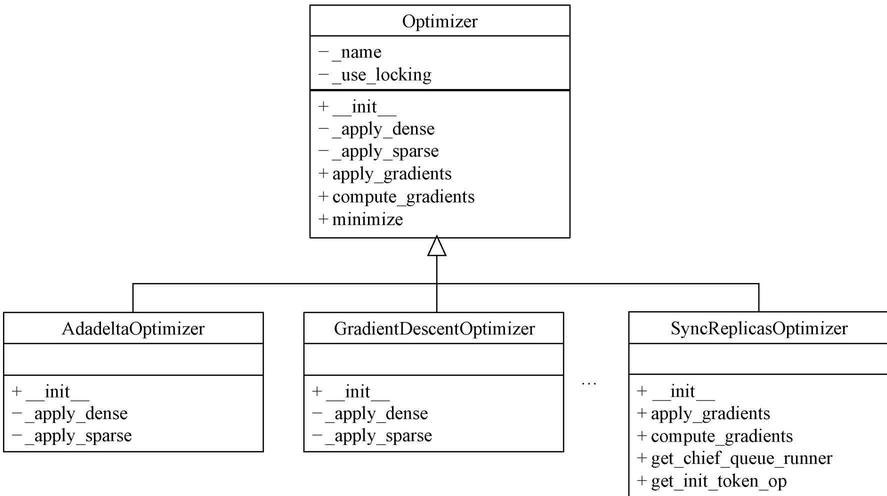


图 3-16 `Optimizer` 类与其子类的继承关系

图 3-16 只列出了 Optimizer 类常见的成员变量和成员方法。_name 属性为字符串型，表示优化器的名称；_use_locking 属性为布尔型，表示是否在并发更新模型参数时加锁。Optimizer 类已经实现了 minimize、compute_gradients 和 apply_gradients 等常用的顶层方法，其子类一般直接继承这 3 个方法。但子类必须实现各自的 _apply_dense 和 _apply_sparse 方法，它们分别表示使用稠密梯度值和稀疏梯度值更新模型参数的具体实现，两者均返回数据流图上的操作。

3.5.3　使用minimize方法训练模型

模型训练的过程需要最小化损失函数。为了方便用户快速上手，TensorFlow 的所有优化器均实现了用于最小化损失函数的 minimize 方法。该方法在内部会依次调用优化器的 compute_gradients 和 apply_gradients 方法。前者计算模型所有参数的梯度值，后者将梯度值更新到对应的模型参数。minimize 方法的代码如下所示，这里我们对其中的关键步骤加以注释：

```
def minimize(self, loss, global_step=None, var_list=None,
             gate_gradients=GATE_OP, aggregation_method=None,
             colocate_gradients_with_ops=False, name=None,
             grad_loss=None):
  # 计算梯度，得到组合后的梯度值与模型参数列表——grads_and_vars，
  # 即<梯度，参数>键值对列表
  grads_and_vars = self.compute_gradients(
      loss, var_list=var_list, gate_gradients=gate_gradients,
      aggregation_method=aggregation_method,
      colocate_gradients_with_ops=colocate_gradients_with_ops,
      grad_loss=grad_loss)
  # 从grads_and_vars中取出非零梯度值对应的模型参数列表——vars_with_grad
  vars_with_grad = [v for g, v in grads_and_vars if g is not None]
  # 如果没有非零梯度值，则说明模型计算过程中出现了问题
  if not vars_with_grad:
    raise ValueError(
        "No gradients provided for any variable, check your graph for ops"
        " that do not support gradients, between variables %s and loss %s." %
        ([str(v) for _, v in grads_and_vars], loss))
  # 使用非零梯度值更新对应的模型参数
  return self.apply_gradients(grads_and_vars, global_step=global_step,
                              name=name)
```

表 3-16 简单介绍了 minimize 方法的所有输入参数。其中，global_step 表示全局训练步数。因为模型训练是一个不断输入数据进行迭代优化的过程，所以我们通常将填充数据、计算梯度、应用梯度更新模型参数这个过程称为一步训练。为了查看训练过程中各评价指标和模型参数的历史变化，需要以训练步数为单位来区分每一步训练结果。虽然 global_step 也保存在变量中，但是它不需要通过输入数据来进行训练。因此，在创建 global_step 变量时，需要显式地将 trainable 参数设置为 False，表示不需要在训练过程中自动计算其梯度值。minimize 方法内部在应用梯度成功后，会将 global_step 变量的值加 1。通常，我们将 minimize 方法的返回操作命名为 train_op 或 train_step，成功执行此操作就表示模型完成了一步训练。

表 3-16 minimize 方法的输入参数

参数名称	功能说明	数据类型
loss	损失值	Tensor
global_step	全局训练步数，随着模型迭代优化自增	Variable
var_list	待训练模型参数的列表	list
gate_gradients	计算梯度和更新参数模型时的并行化程度，可选值为 GATE_OP（默认值）、GATE_NONE 或 GATE_GRAPH	Enum
aggregation_method	聚集梯度值的方法，可选值定义在 AggregatedMethod 类中	Enum
colocate_gradients_with_ops	是否将梯度计算放置到对应操作所在的同一个设备，默认为否	Boolean
name	优化器在数据流图中的名称	String
grad_loss	损失值的梯度	Tensor

在上述参数中，gate_gradients 对计算效率有一定影响，值得用户重点关注。用户可以根据自身需求设置 gate_gradients 的取值，它的三种不同取值的含义如下。

- GATE_NONE：无同步。最大化并行执行效率，将梯度计算和模型参数更新过程完全并行化。该模式有时会导致部分计算结果无法复现。
- GATE_OP：操作级同步。对于每个操作，分别确保所有梯度在使用之前都已计算完成。当梯度计算依赖多个输入时，这种做法能够避免计算间的竞争。
- GATE_GRAPH：图级同步。最小化并行执行效率，在任意梯度被使用之前，确保所有模型参数对应的所有梯度都已经计算完成。

最佳实践

无论模型如何设计，最终都需要使用优化器来进行训练。下面我们以梯度下降优化器为例，介绍使用 minimize 方法训练模型的典型步骤。代码如下所示：

```
import tensorflow as tf
# 模型
X = tf.placeholder(...)
Y_ = tf.placeholder(...)
w = tf.Variable(...)
b = tf.Variable(...)
Y = tf.matmul(X, w) + b
# 使用交叉熵作为损失函数
loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=Y_, logits=Y))
# 优化器
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
global_step = tf.Variable(0, name='global_step', trainable=False)
train_op = optimizer.minimize(loss, global_step=global_step)
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  for step in xrange(max_train_steps):
    sess.run(train_op, feed_dict={...})
```

```
    # 训练日志
    if step % log_steps == 0:
      final_loss, weight, bias = sess.run([loss, w, b],
                                          feed_dict={...})
      print("Step: %d, loss==%.4f, w==%.4f, b==%.4f",
            step, final_loss, weight, bias)
    ...
```

首先，我们需要创建模型和定义损失函数，并得到对应的损失值（loss）。然后，调用 GradientDescentOptimizer 类的构造方法，传入学习率（learning_rate）参数，创建梯度下降优化器实例 optimizer。接着，创建记录全局训练步数的 global_step 变量，在构造函数中设置其名称和不可训练的属性。在此基础上。调用 optimizer 对象的 minimize 方法，并传入损失值（loss）和 global_step 变量，以便创建单步训练操作（train_op）。在完成所有变量定义之后，我们需要创建会话，并初始化所有变量值。最后，通过不断执行单步训练操作来训练模型。这里可以设置一个最大训练步数或训练中止条件（如准确率达到 90% 或损失值低于 0.1 等）。同时，也可以定期地打印输出训练日志，以便于观察模型的训练效果。

3.5.4　扩展阅读：模型训练方法进阶

让我们复盘模型训练的最关键部分——优化算法。上一节介绍了使用 minimize 方法训练模型的最佳实践，该方法具有简单易用的特点，但无法实现对梯度进行特定处理的需求。通过阅读 minimize 方法的源代码，我们发现它内部先后调用了 compute_gradients 和 apply_gradients 方法，而对梯度进行处理的逻辑恰恰可以位于这两者之间。因此，一次更为通用的迭代优化过程可以分为以下三个步骤。

(1) 计算梯度：调用 compute_gradients 方法，依据指定的策略求得梯度值。

(2) 处理梯度：用户按照自己的需求处理梯度值，如进行梯度裁剪和梯度加权。

(3) 应用梯度：调用 apply_gradients 方法，将处理后的梯度值应用到模型参数，实现模型更新。

下面分别介绍这三个步骤的实现原理。

1. 计算梯度

TensorFlow 的所有原生优化器都使用基类 Optimizer 的 compute_gradients 方法计算梯度，该方法的代码及关键步骤解释如下：

```
def compute_gradients(self, loss, var_list=None,
                      gate_gradients=GATE_OP,
                      aggregation_method=None,
                      colocate_gradients_with_ops=False,
                      grad_loss=None):
  if gate_gradients not in [Optimizer.GATE_NONE, Optimizer.GATE_OP,
                            Optimizer.GATE_GRAPH]:
    raise ValueError("gate_gradients must be one of: Optimizer.GATE_NONE, "
```

```
                    "Optimizer.GATE_OP, Optimizer.GATE_GRAPH.  Not %s" %
                    gate_gradients)
  self._assert_valid_dtypes([loss])
  if grad_loss is not None:
    self._assert_valid_dtypes([grad_loss])
  if var_list is None:
    var_list = (
        variables.trainable_variables() +
        ops.get_collection(ops.GraphKeys.TRAINABLE_RESOURCE_VARIABLES))
  processors = [_get_processor(v) for v in var_list]
  if not var_list:
    raise ValueError("No variables to optimize.")
  var_refs = [p.target() for p in processors]
  # 创建计算梯度的操作
  grads = gradients.gradients(
      loss, var_refs, grad_ys=grad_loss,
      gate_gradients=(gate_gradients == Optimizer.GATE_OP),
      aggregation_method=aggregation_method,
      colocate_gradients_with_ops=colocate_gradients_with_ops)
  # 如果采用 GATE_GRAPH 策略，则为所有计算梯度的操作添加控制依赖边，
  # 以保证所有梯度值在使用前都已经计算完毕
  if gate_gradients == Optimizer.GATE_GRAPH:
    grads = control_flow_ops.tuple(grads)
  # 创建梯度和模型参数表
  grads_and_vars = list(zip(grads, var_list))
  self._assert_valid_dtypes([v for g, v in grads_and_vars if g is not None])
  return grads_and_vars
```

在 `compute_gradients` 方法的内部实现中，值得重点关注的是 `gradients.gradients` 方法。它的主要输入是损失值和模型参数，输出是计算输入模型参数对应梯度值的操作（`grads`）。这个操作实现了梯度计算，它能够返回一个包含所有变量对应梯度值的张量列表。`gradients.gradients` 方法对应图 3-15 中的 gradients 子图，感兴趣的读者可以在 tensorflow/python/ops/gradients_impl.py 文件中找到它的定义。该方法的原型如下所示：

```
def gradients(ys,
              xs,
              grad_ys=None,
              name="gradients",
              colocate_gradients_with_ops=False,
              gate_gradients=False,
              aggregation_method=None)
```

计算出梯度后，`compute_gradients` 方法内部会创建一个梯度和模型参数表，表中便是待更新的模型参数与对应的梯度值。此表会作为方法返回值，提供给用户进行下一步的梯度处理工作。

2. 处理梯度

通常，我们在训练模型时总希望快速收敛，但实际情况却往往不那么顺利，梯度有可能在训练过程中变成了无限大或者零值。出现这种情况时，有可能是因为输入数据本身不合法，也有可能是除法或求导运算的精度限制导致的。换句话说，即使模型设计没有问题，仍然存在训练无法顺利进行的情况。为了回避梯度爆炸或梯度消失问题，我们可以对梯度值进行一些处理，再更新

到模型参数。例如，可以实施梯度裁剪、梯度加权、模型平均（model averaging）和批规范化（batch normalization）等。

TensorFlow 为用户提供了许多内置的梯度处理方法，以便用户快速实现常见的梯度处理需求。表 3-17 给出了这些方法的功能说明。同时，TensorFlow 也支持自定义梯度处理操作，以此来支持更加个性化的需求。

表 3-17　TensorFlow 内置的梯度处理方法

方　　法	功能说明
`tf.clip_by_value(t, clip_value_min, clip_value_max, name=None)`	将梯度值 `t` 截断到`[min, max]`区间内，确保梯度的最大值和最小值分别为 `max` 和 `min`
`tf.clip_by_norm(t, clip_norm, axes=None, name=None)`	使用 L2 范式规范化梯度 `t` 的最大值为 `clip_norm`，返回 `t * clip_norm / l2norm(t)`
`tf.clip_by_average_norm(t, clip_norm, name=None)`	使用平均 L2 范式规范化梯度 `t` 的最大值为 `clip_norm`，返回 `t * clip_norm / l2norm_avg(t)`
`tf.clip_by_global_norm(t_list,clip_norm, use_norm=None, name=None)`	根据全局规范化的梯度值列表 `t_list` 的加和进行裁剪，返回 `t_list[i] * clip_norm / max(global_norm, clip_norm)`
`tf.global_norm(t_list, name=None)`	计算 `t_list` 中所有梯度的全局范数，返回 `global_norm = sqrt(sum([l2norm(t)**2 for t in t_list]))`

下面我们以使用 `tf.clip_by_norm` 方法对 Adam 优化器计算出的梯度值进行裁剪为例，介绍 TensorFlow 中梯度处理的典型步骤。代码如下所示：

```
import tensorflow as tf
... # 创建模型
optimizer = tf.train.AdamOptimizer(learning_rate, beta1=0.5)
grads_and_vars = optimizer.compute_gradients(loss)
for i, (g, v) in enumerate(grads_and_vars):
  if g is not None:
    grads_and_vars[i] = (tf.clip_by_norm(g, 5), v)  # 裁剪梯度
train_op = optimizer.apply_gradients(grads_and_vars)
... # 创建会话，训练模型
```

在本例中，创建模型、定义 Adam 优化器及计算梯度的方法参考了上一节的最佳实践。取得梯度值和模型参数列表 `grads_and_vars` 之后，可以使用 `tf.clip_by_norm` 方法对列表中所有非零梯度值进行裁剪和更新，以实现 L2 范式规范化。在参数更新之后，我们调用 Adam 优化器的 `apply_gradients` 方法，传入新的梯度值和模型参数列表。后续会话创建、模型训练的步骤同前例，并无差异。

3. 应用梯度

作为 `minimize` 方法内部实现的后半部分，`apply_gradients` 方法的主要输入是处理后的梯度值与对应模型参数组合而成的列表（`grads_and_vars`），输出是用于更新所有模型参数的操作（`apply_udpates`）。下面给出了基类 `Optimizer` 中 `apply_gradients` 方法的定义，我们结合代码分析它的工作原理：

```
def apply_gradients(self, grads_and_vars, global_step=None, name=None):
  grads_and_vars = tuple(grads_and_vars)
  if not grads_and_vars:
    raise ValueError("No variables provided.")
  converted_grads_and_vars = []
  for g, v in grads_and_vars:
    if g is not None:
      try:
        g = ops.convert_to_tensor_or_indexed_slices(g)
      except TypeError:
        raise TypeError(
            "Gradient must be convertible to a Tensor"
            " or IndexedSlices, or None: %s" % g)
      if not isinstance(g, (ops.Tensor, ops.IndexedSlices)):
        raise TypeError(
            "Gradient must be a Tensor, IndexedSlices, or None: %s" % g)
    p = _get_processor(v)
    converted_grads_and_vars.append((g, v, p))
  converted_grads_and_vars = tuple(converted_grads_and_vars)
  var_list = [v for g, v, _ in converted_grads_and_vars if g is not None]
  if not var_list:
    raise ValueError("No gradients provided for any variable: %s." %
                     ([str(v) for _, _, v in converted_grads_and_vars],))
  with ops.control_dependencies(None):
    self._create_slots(var_list)

  update_ops = []
  with ops.name_scope(name, self._name) as name:
    self._prepare()
    for grad, var, processor in converted_grads_and_vars:
      if grad is None:
        continue
      with ops.name_scope("update_" + var.op.name), ops.colocate_with(var):
        update_ops.append(processor.update_op(self, grad))

    if global_step is None:
      apply_updates = self._finish(update_ops, name)
    else:
      with ops.control_dependencies([self._finish(update_ops, "update")]):
        with ops.colocate_with(global_step):
          apply_updates = state_ops.assign_add(global_step, 1, name=name).op

    train_op = ops.get_collection_ref(ops.GraphKeys.TRAIN_OP)
    if apply_updates not in train_op:
      train_op.append(apply_updates)

    return apply_updates
```

apply_gradients 方法接受经过处理的梯度和模型参数列表 grads_and_vars 作为输入。处理后的梯度值可能不是张量类型了，但是程序仍然需要使用张量类型来表示梯度，以此来继续构建数据流图。为此，apply_gradients 方法内部首先需要将梯度值转换为张量类型。如果能够成功转换，那么将其与对应的模型参数保存到 converted_grads_and_vars 列表。随后，将

converted_grads_and_vars 列表转换为元组类型，使其能够支持列表推导式运算。

接着，apply_gradients 方法逐对遍历 converted_grads_and_vars 中的梯度值和模型参数，根据产生模型参数的操作名称找到对应参数的更新操作，并将其追加到更新操作汇总表 update_ops 中。在汇总所有的更新操作后，使用 Optimizer 的 _finish 方法增加显式的控制依赖。根据数据流图中的拓扑结构，程序会先执行 apply_updates 依赖的所有输入节点，即更新操作汇总表 update_ops 中的所有更新操作。在并行更新模型参数的过程中，Optimizer._finish 方法能够确保更新操作按照拓扑序正确执行。它的定义如下：

```
def _finish(self, update_ops, name_scope):
  return control_flow_ops.group(*update_ops, name=name_scope)
```

换句话说，在 apply_updates 操作执行之前，update_ops 中的所有操作都已经执行完成，以确保所有模型参数都已更新成功。如果 global_step 变量不为空，执行完 update_ops 中的所有操作后，global_step 变量值会被加 1，以此作为训练步数的记录。最后，apply_updates 成为单步训练的计算节点。即当用户成功执行一次 apply_updates 操作时，模型就完成了一步训练。

其中，模型参数更新操作在内部调用了优化器的 _apply_dense 或 _apply_sparse 方法，这两个方法分别实现了使用稠密梯度值和稀疏梯度值更新对应模型参数的逻辑。每种优化器的更新策略千差万别，基类 Optimizer 无法实现通用的更新策略。因此，它只是简单地定义了这两个成员方法，并为其添加了 NotImplementedError 异常，以确保子类在继承时必须实现相应的方法。如果用户想要自定义优化器，则需要分别实现这两个成员方法。

综上，模型训练的关键是对优化器的学习和掌握。本节中，我们从理论指导和实践落地这两个角度分别介绍了 TensorFlow 的训练工具——优化器。在使用优化器训练模型时，我们可以选择简单易用的 minimize 方法，也可以调用 compute_gradients 和 apply_gradients 方法，并插入梯度处理逻辑来增加灵活性。

3.6　一元线性回归模型的最佳实践

本节以最佳实践的形式介绍如何利用张量、操作、优化器和会话创建并训练简单的一元线性回归模型。读者通过详细阅读各核心步骤的典型实现，结合前面 5 节的知识，能够更加深入地理解 TensorFlow 的基本概念和使用方法。

在统计学中，线性回归（linear regression）是一种回归分析方法，它利用基于最小二乘函数的回归方程对一个或多个自变量和因变量之间的关系进行建模。只有一个自变量的情况称为简单回归，大于一个自变量的情况叫作多元回归或 N 元回归。一般情况下，具有 N 个自变量的 N 元线性回归模型的形式化定义如下：

$$Y = \boldsymbol{W}^{\mathrm{T}}X + b = w_1x_1 + w_2x_2 + \cdots + w_Nx_N + b$$

其中，$\boldsymbol{W}$ 是权重矩阵，b 是偏置量，X 是自变量，Y 是因变量。N 元线性回归模型描述了 N 个自变量与 1 个因变量之间的线性关系。当 $N = 1$ 时，即：

$$Y = \boldsymbol{W}X + b = w_1x_1 + b$$

图 3-17 展示了一元线性回归模型的数据流图表达形式。其中，前向图的结构是权重矩阵 $\boldsymbol{W}$ 与因变量 X 相乘，再加上偏置 b，得到推理值 Y；后向图则基于梯度下降优化算法，使用优化器计算出的梯度值直接更新权重矩阵 $\boldsymbol{W}$ 和偏置 b。

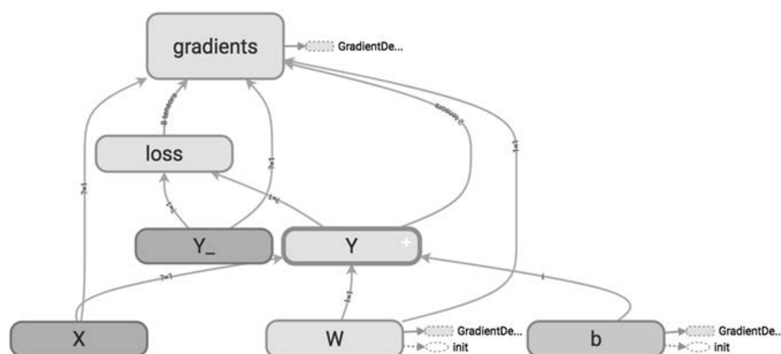


图 3-17 一元线性回归模型的数据流图表达形式

使用 TensorFlow 训练该模型的典型过程可以分为以下 8 个步骤：

(1) 定义超参数；

(2) 输入数据；

(3) 构建模型；

(4) 定义损失函数；

(5) 创建优化器；

(6) 定义单步训练操作；

(7) 创建会话；

(8) 迭代训练。

下面分别介绍每个步骤，然后给出可视化模型训练过程的方法供读者参考。

1. 定义超参数

超参数是指模型训练过程中使用的配置参数。超参数的取值往往影响模型的收敛速度或预测精度。常见的超参数有学习率、隐藏层神经元个数、批数据个数和正则项系数等。通常情况下，我们会参考经典模型训练的经验数值设置超参数，比如将梯度下降优化器的学习率设为 0.01：

```
# 学习率
learning_rate = 0.01
# 最大训练步数
max_train_steps = 1000
```

2. 输入数据

在有监督的机器学习中，数据集一般划分为训练集（training set）、验证集（validation set）和测试集（test set）。通常，训练集用于训练模型或确定模型参数；验证集用于控制模型复杂度或确定模型结构；测试集用于测试模型的性能和泛化能力。本例构造了 17 对训练数据：

```
# 构造训练数据
train_X = np.array([[3.3],[4.4],[5.5],[6.71],[6.93],[4.168],[9.779],[6.182],[7.59],[2.167],
    [7.042],[10.791],[5.313],[7.997],[5.654],[9.27],[3.1]], dtype=np.float32)
train_Y = np.array([[1.7],[2.76],[2.09],[3.19],[1.694],[1.573],[3.366],[2.596],[2.53],
    [1.221],[2.827],[3.465],[1.65],[2.904],[2.42],[2.94],[1.3]], dtype=np.float32)
total_samples = train_X.shape[0]
```

因为一元线性回归模型的结构非常简单，所以不再单独构造验证集和测试集。

3. 构建模型

构建模型是指编写数据流图对应的代码，描述算法模型结构的过程。对于训练数据的属性，可以使用占位符描述。本例中的 X 是单精度浮点数，形状定义为 [None, 1]，其中 None 表示支持任意个数的输入，1 表示数据是一维的。对于变量和计算操作，则使用相应的节点构造方法或全局函数定义。模型参数 W 是形状为 [1, 1] 的单精度浮点数矩阵，偏置量 b 是单精度浮点数标量，它们分别使用正态分布和零值进行初始化。将 X 和 W 都定义为矩阵，是为了方便将所有的训练数据一次性填充以进行推理计算。推理值 Y 的定义源于一元线性回归模型的形式化定义。需要注意的是，这行语句属于声明式的、用于构图的符号表达式，而非命令式的、立即执行的代码。相关代码如下：

```
# 输入数据
X = tf.placeholder(tf.float32, [None, 1])
# 模型参数
W = tf.Variable(tf.random_normal([1, 1]), name="weight")
b = tf.Variable(tf.zeros([1]), name="bias")
# 推理值
Y = tf.matmul(X, W) + b
```

4. 定义损失函数

接着，我们使用 $Y_$ 描述训练数据 X 对应的实际值。这里采用均方差作为模型训练的损失函数。均方差具有形式简单、计算量小等特点，它的形式化定义如下：

$$\text{loss} = \text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - Y_{-i})^2$$

相关代码如下：

```
# 实际值
Y_ = tf.placeholder(tf.float32, [None, 1])
# 均方差
loss = tf.reduce_sum(tf.pow(Y-Y_, 2))/(total_samples)
```

5. 创建优化器

一元线性回归模型比较简单，本例直接使用随机梯度下降优化器计算梯度值：

```
# 随机梯度下降
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
```

6. 定义单步训练操作

这里将单步训练操作（train_op）设置为使用随机梯度下降优化器的 minimize 方法计算和应用梯度。minimize 方法在内部实现了梯度计算和模型更新的过程。每当用户在会话中成功执行一次单步训练操作，就使得前向图和后向图依次执行了一遍。相关代码如下：

```
# 最小化损失值
train_op = optimizer.minimize(loss)
```

7. 创建会话

定义好算法模型相关的对象之后，需要定义 TensorFlow 运行时框架相关的对象。对于本例这种简单的程序，只需要定义用于维护训练上下文的会话对象。这里使用 Session 类的构造方法创建会话，然后调用 global_variables_initializer 方法初始化全局变量：

```
with tf.Session() as sess:
  # 初始化全局变量
  sess.run(tf.global_variables_initializer())
```

8. 迭代训练

前面我们已经设定模型总共迭代训练 1000 步。每一步训练都需要填充全部 17 对训练数据。因为模型参数 W 和 b 会不断更新，所以同样的训练数据集上每一步计算出的损失值都不同。在示例代码中，我们让程序定期输出日志，以便观察参数和损失值的变化。训练结束之后，计算并输出最终的模型参数。相关代码如下：

```
print("Start training:")
for step in xrange(max_train_steps):
  sess.run(train_op, feed_dict={X: train_X, Y_: train_Y})
  # 每隔 log_step 步打印一次日志
  if step % log_step == 0:
    c = sess.run(loss, feed_dict={X: train_X, Y_:train_Y})
    print("Step:%d, loss==%.4f, W==%.4f, b==%.4f" %
         (step, c, sess.run(W), sess.run(b)))
# 计算训练完毕的模型在训练集上的损失值，并将其作为指标输出
final_loss = sess.run(loss, feed_dict={X: train_X, Y_: train_Y})
# 计算训练完毕的模型参数 W 和 b
weight, bias = sess.run([W, b])
print("Step:%d, loss==%.4f, W==%.4f, b==%.4f" %
     (max_train_steps, final_loss, sess.run(W), sess.run(b)))
print("Linear Regression Model: Y==%.4f*X+%.4f" % (weight, bias))
```

9. 模型可视化

在 Jupyter Notebook 等 Python 交互式环境下，我们可以使用 matplotlib 库实现模型的可视化。

具体步骤为：首先，执行 %matplotlib 命令，初始化 matplotlib 后端。然后，调用 plt.plot 方法，设置横纵坐标的指标和属性。最后，调用 plt.show 方法绘制图形。相关代码如下：

```
# 初始化 matplotlib 后端
%matplotlib
# 根据训练数据 X 和 Y，添加对应的红色圆点
plt.plot(train_X, train_Y, 'ro', label='Training data')
# 根据模型参数和训练数据，添加蓝色（默认色）拟合直线
plt.plot(train_X, weight * train_X + bias, label='Fitted line')
# 添加图例说明
plt.legend()
# 绘制图形
plt.show()
```

输出结果如图 3-18 所示。

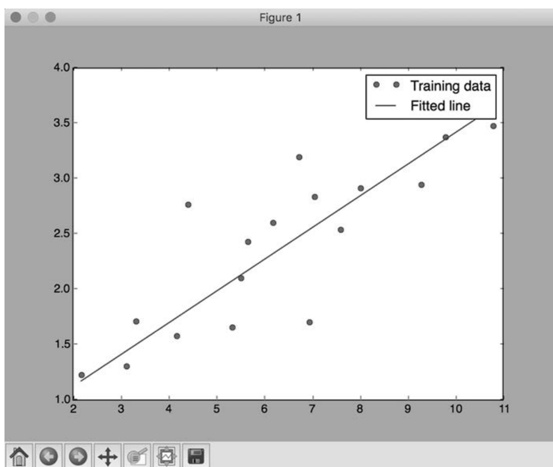


图 3-18　使用 matplotlib 渲染的一元线性回归模型

3.7　小结

TensorFlow 的核心编程范式是基于声明式编程的数据流图。数据流图由节点和有向边组成，能够有效表达机器学习和深度学习领域的各类算法模型，具有代码可读性强、支持引用透明、提供预编译优化能力等优点。从实现的角度来看，一个典型的 TensorFlow 应用程序包含数据载体、模型载体、运行环境和训练工具这 4 个部分。张量是 TensorFlow 的数据载体，它可以描述数据流图上流动的任意类型和形状的数据，具有极高的灵活性。操作是 TensorFlow 的模型载体，计算操作、变量操作和占位符操作分别描述了模型的计算拓扑、模型参数和数据集。会话是 TensorFlow 的运行环境，它连接后端执行引擎，为数据流图的执行维护计算资源和存储资源。优化器是 TensorFlow 的训练工具，它为用户提供了多种梯度计算和模型更新的方法。掌握这些基础概念、习惯声明式编程思维，是正确地使用 TensorFlow 开发应用的前提。

第二部分

关键模块篇

第 4 章 TensorFlow 数据处理方法

神经网络的复兴和深度学习的崛起离不开大数据的推动。对于我们这个时代的机器学习模型，数据是一切的基础。没有真实而有效的数据支撑，再精妙的算法也会黯然失色，模型设计便如同纸上谈兵。因此，读者在学习 TensorFlow 模型开发方法的同时，很有必要掌握其数据处理方法。TensorFlow 主要涉及三类数据，分别是输入数据集、模型参数和命令行参数。其中，输入数据集指用于模型训练、验证和测试的数据集合。模型参数主要指模型算法涉及的权重值和偏置值等。命令行参数则指用户启动 TensorFlow 程序时输入的可选项，包含模型超参数和集群参数等。本章将依次介绍以上三类数据的处理流程，具体包括数据的输入、使用、输出和存储等方法，旨在帮助读者理解 TensorFlow 数据处理模块的用法和原理，从而使得开发出的应用更具实用价值。

4.1 输入数据集

用户处理输入数据集的典型流程是：首先将输入数据集从文件系统读取到内存中，然后将其转换为模型需要的输入数据格式，接着以某种方式传入数据流图，继而开始真正的模型训练过程。输入数据集一般被存储在各种类型的文件系统中，如本地文件系统、共享文件系统和分布式文件系统等。根据文件系统类型和输入数据集的大小，我们推荐如下两种不同的数据读取方法。

- 大数据集（如 ImageNet）一般由大量数据文件组成。因为数据规模太大，所以无法一次性全部加载到内存。于是，用户只能在每一步训练时加载数据，这将阻塞模型的计算任务。为了减小数据读取对模型训练效率的影响，我们可以使用多线程并行地读取数据和训练模型。
- 小数据集（如 MNIST）可能仅包含一个文件，因此用户可以在模型训练开始前一次性将其加载到内存处理。本节依次讲解并行读取数据、创建批样例数据，以及填充数据节点的方法，然后介绍 CIFAR-10 和 MNIST 数据集及其用例。

4.1.1 使用输入流水线并行读取数据

大数据集一般指无法一次性加载到内存中进行处理的数据集，如 ImageNet 图像分类数据集

（约为 140GB）。当处理如此规模的大数据集时，TensorFlow 提供了以输入流水线方式从多个文件中并行读取数据的方法，这使得模型训练所需的数据能够实时填充进数据流图。该方法的核心思想是实现多个数据缓冲区以确保任何时刻内存中都有数据可以填充进数据流图。图 4-1 展示了一个典型的输入流水线并行读取数据的工作流程，我们认为可以将该流程分为以下 4 个关键步骤：

(1) 创建文件名列表；

(2) 创建文件名队列；

(3) 创建 Reader 和 Decoder；

(4) 创建样例队列。

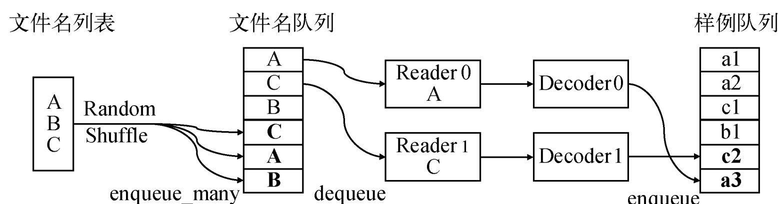


图 4-1 使用输入流水线并行读取数据的工作流程

理解整个工作流程的关键是理解两个队列：文件名队列和样例队列。因为模型训练过程不止一次遍历整个数据集，所以文件名队列为程序读取数据文件提供了一个缓冲区。在将文件名传入文件名队列时，程序打乱了文件名的顺序，增加了输入数据的随机性。又因为程序需要向数据流图中持续不断地填充符合特定数据属性的样例，所以样例队列为填充数据流图提供了一个缓冲区，使得每一步训练都能够实时地获取到输入样例。下面介绍每一步流程的典型实现方法。

1. 创建文件名列表

文件名列表是指组成输入数据集的所有文件的名称构成的列表，它们可能是本地文件系统上的文件位置，也可能是共享文件系统或分布式文件系统上的统一资源标志符（URI）。用户需要确保 TensorFlow 程序有权限访问 URI 标识的文件。这里我们推荐下面两种创建文件名列表的方法。

- **使用 Python 列表**。如果文件名的个数不多，或文件命名遵循某种规则，那么用户可以直接使用 Python 列表存储文件名，比如 `["file0.csv", "file1.csv"]` 或者 `[("file%d.csv" % i) for i in xrange(100)]`。
- **使用 `tf.train.match_filenames_once` 方法**。该方法在数据流图中创建一个获取文件名列表的操作，它输入一个文件名列表的匹配模式，返回一个存储了符合该匹配模式的文件名列表变量。在初始化全局变量时，该文件名列表变量也会被初始化。

2. 创建文件名队列

我们使用 `tf.train.string_input_producer` 方法创建文件名队列，它的输入是前面创建的文件名列表，输出是一个先入先出（FIFO）的文件名队列。通常，我们称完整遍历一次输入数

据集为模型的一个训练周期，而训练模型需要反复遍历整个输入数据集，以不断检验更新的模型参数是否能够更好地表达训练数据的潜在模式。用户可以通过 tf.train.string_input_producer 方法的输入参数 num_epochs 设置模型的最大训练周期数。但是在每一次遍历数据集时，我们希望输入样本的顺序有所不同，通过增加一些随机因素减小模型的过拟合。因此，我们可以将 tf.train.string_input_producer 方法的输入参数 shuffle 设置为 True，此时程序便能打乱每个训练周期的文件名顺序。同时，TensorFlow 保证打乱文件名顺序后仍然采用均匀抽样，避免了用户自己实现文件名乱序时可能造成的欠采样或过采样的问题。如图 4-1 所示，文件名列表中的文件名顺序原本是 ABC，乱序后输入文件名队列中的顺序是 ACB 和 CAB。tf.train.string_input_producer 方法的原型如下：

```
tf.train.string_input_producer(string_tensor, num_epochs=None, shuffle=True, seed=None,
    capacity=32, shared_name=None, name=None, cancel_op=None)
```

表 4-1 列出了该方法的所有输入参数，它们均可以作为创建文件名时的配置项。

表 4-1　tf.train.string_input_producer 方法的输入参数

参数名称	功能说明
string_tensor	存储文件名列表的字符串张量
num_epochs	最大训练周期
shuffle	是否打乱文件名顺序
seed	随机化种子，当 shuffle 等于 True 的时候生效
capacity	文件名队列容量
shared_name	多个会话间共享的文件名队列名称
name	创建文件名队列操作的名称
cancel_op	取消队列的操作

3. 创建 Reader 和 Decoder

Reader 的功能是读取数据记录，Decoder 的功能是将数据记录转换为张量格式。Reader 和 Decoder 的类型与数据文件格式相关，表 4-2 列出了 TensorFlow 推荐的 3 种数据文件格式及其对应的 Reader 和 Decoder 类型。我们使用 Reader 和 Decoder 的典型流程是：首先，创建输入数据文件对应的 Reader。然后，从文件名队列中取出文件名。接着，将它传入 Reader 的 read 方法，后者返回形如 (输入数据文件, 数据记录) 的元组。最后，使用对应的 Decoder 操作，将数据记录中的每一列数据都转换为张量格式。

表 4-2　TensorFlow 推荐的 3 种数据文件格式

文件格式	Reader 类型	Decoder 类型
CSV 文件	tf.TextLineReader	tf.decode_csv
TFRecords 文件	tf.TFRecordReader	tf.parse_single_example
自由格式文件	tf.FixedLengthRecordReader	tf.decode_raw

下面分别介绍这 3 种格式数据文件的典型处理流程。

- **CSV 文件**

字符分隔值（Comma-Separated Values，CSV）文件以纯文本形式存储表格数据。虽然 RFC 4180 对 CSV 文件有一些基础性描述，但是 CSV 并没有通用的法定标准。不过这并不妨碍 CSV 文件事实标准的形成。目前被大部分人接受的标准是 CSV 文件由多条数据记录组成，记录间以某种换行符分隔。每条记录由多个字段组成，字段间通常以制表符或逗号分隔。所有记录拥有完全相同的字段序列。许多数据库都以 CSV 文件格式作为存储数据的文件格式。假设我们需要读取一张记录收入支出的表格数据，各字段如表 4-3 所示，存储该表格数据的文件为 stat0.csv 和 stat1.csv。

表 4-3 记录收入支出的表格数据示例

id	age	income	outgo
1	24	2048.0	1024.0
2	48	4096.0	2048.0

对于 stat0.csv 和 stat1.csv 文件，一条数据记录包含 4 个字段——`id`、`age`、`income` 和 `outgo`，其中字段间有分隔符严格区分。下面看看如何读取上述 CSV 文件中的数据：

```
# 创建文件名队列 filename_queue
filename_queue = tf.train.string_input_producer(['stat0.csv', 'stat1.csv'])
# 创建读取 CSV 文件的 TextLineReader
reader = tf.TextLineReader()
# 从文件名队列中取出 CSV 文件中的一条数据记录 value
_, value = reader.read(filename_queue)
# 设置数据记录的默认值
record_defaults = [[0], [0], [0.0], [0.0]]
# 使用 decode_csv 方法将数据记录的每个字段都转换为特征张量
id, age, income, outgo = tf.decode_csv(value,
                                       record_defaults=record_defaults)
# 将所有特征张量组合在一起形成一条记录
features = tf.stack([id, age, income, outgo])
```

除了使用 `read` 方法每次读取一行数据记录外，还可以使用 `read_up_to` 方法一次读取多条数据。它的原型是 `read_up_to(queue, num_records, name=None)`，通过设置 `num_records` 参数可以显式指定程序一次读取的数据记录数量。此外，我们为 `decode_csv` 方法设置了 `record_defaults` 参数，它的作用是在某些字段数据不合法或不存在时，为该字段填充 `record_defaults` 中定义的默认值，确保程序能够继续正确执行。需要注意的是，`read` 方法和 `decode_csv` 方法返回的都是数据流图上的操作，而不是真实的数据。因此，用户需要在会话中执行上述操作才能获取到输入数据。

- **TFRecords 文件**

TFRecords 文件存储的是有结构的序列化字符块，它是 TensorFlow 推荐的标准文件格式。TensorFlow 通过 Protocol Buffers 定义了 TFRecords 文件中存储的数据记录及其所含字段的数据结构，它们分别定义在 tensorflow/core/example 目录下的 example.proto 和 feature.proto 文件中。因

此，我们将数据记录转换后的张量称为样例，将记录包含的字段称为特征域。TFRecords 文件的样例结构层次非常清晰，一个样例包含一组特征。一组特征由多个特征向量组成的 Python 字典构成。为了说明读取 TFRecords 文件中样例的方法，我们首先使用 `tf.python_io.TFRecordWriter` 方法将表 4-3 中的数据写入 TFRecords 文件 stat.tfrecord 中，示例如下：

```
'''writer.py'''
import tensorflow as tf
# 创建向 TFRecords 文件写数据记录的 writer
writer = tf.python_io.TFRecordWriter('stat.tfrecord')
# 两轮循环构造输入样例
for i in range(1,3):
  # 创建 example.proto 中定义的样例
  example = tf.train.Example(
      features = tf.train.Features(
          feature = {
            'id': tf.train.Feature(int64_list =
                tf.train.Int64List(value=[i])),
            'age': tf.train.Feature(int64_list =
                tf.train.Int64List(value=[i*24])),
            'income': tf.train.Feature(float_list =
                tf.train.FloatList(value=[i*2048.0])),
            'outgo': tf.train.Feature(float_list =
                tf.train.FloatList(value=[i*1024.0]))
          }
      )
  )
  # 将样例序列化为字符串后，写入 stat.tfrecord 文件
  writer.write(example.SerializeToString())
# 关闭输出流
writer.close()
```

然后使用 `tf.TFRecordReader` 方法读取 stat.tfrecord 文件中的样例，接着使用 `tf.parse_single_example` 将样例转换为张量。`tf.parse_single_example` 方法的输入参数 `features` 是一个 Python 字典，具体包括组成样例的所有特征的名称和数据类型，它们必须与 writer.py 中使用 `tf.train.Features` 方法定义的特征保持完全一致。`tf.FixedLenFeature` 方法的输入参数为特征形状和特征数据类型。因为本例中的 4 个特征都是标量，所以形状为 `[]`。相关代码如下：

```
'''reader.py'''
import tensorflow as tf
# 创建文件名队列 filename_queue
filename_queue = tf.train.string_input_producer(['stat.tfrecord'])
# 创建读取 TFRecords 文件的 reader
reader = tf.TFRecordReader()
# 取出 stat.tfrecord 文件中的一条序列化的样例 serialized_example
_, serialized_example = reader.read(filename_queue)
# 将一条序列化的样例转换为其包含的所有特征张量
features = tf.parse_single_example(
    serialized_example,
    features={
        'id': tf.FixedLenFeature([], tf.int64),
        'age': tf.FixedLenFeature([], tf.int64),
```

```
        'income': tf.FixedLenFeature([], tf.float32),
        'outgo': tf.FixedLenFeature([], tf.float32),
    }
)
```

- **自由格式文件**

自由格式文件指用户自定义的二进制文件。它的存储对象是字符串，每条数据记录都是一个固定长度的字节块。因此，如果想要正确识别和转换二进制文件中的数据记录，必须首先使用 `tf.FixedLengthRecordReader` 方法读取二进制文件中固定长度的字节块，然后使用 `tf.decode_raw` 方法将读取的字符串转换为 `uint8` 类型的张量，最后按照用户定义的数据结构将这些张量组织为输入样例。`tf.FixedLengthRecordReader` 方法和 `tf.TextLineReader` 方法均继承自 `ReaderBase` 类，它们也支持一次读取多条记录。`tf.decode_raw` 方法的功能是将字符串转换为 `uint8` 类型的张量，表 4-4 介绍了它的输入参数，该方法的原型如下所示：

```
tf.decode_raw(bytes, out_type, little_endian=None, name=None)
```

表 4-4　`tf.decode_raw` 方法的输入参数

参数名称	功能说明
bytes	从二进制文件中读取的字符串
out_type	样例的数据类型，如 tf.half、tf.float32、tf.float64、tf.int32、tf.uint8、tf.int16、tf.int8 和 tf.int64
little_endian	是否使用小端顺序存储数据，默认为 True
name	该方法在数据流图上的操作名称

TFRecords 文件是 TensorFlow 团队定义的自由格式文件，TensorFlow 提供了以 Protocol Buffers 定义的数据结构解析和重构样例的相关方法。读者也可以根据自身需求定义自由格式文件，这里不再举例。

4. 创建样例队列

执行前 3 个步骤后，我们得到 4 个特征张量——`age`、`outgo`、`id` 和 `income`，如下所示：

```
{'age': <tf.Tensor 'ParseSingleExample/Squeeze_age:0' shape=() dtype=int64>,
 'outgo': <tf.Tensor 'ParseSingleExample/Squeeze_outgo:0' shape=() dtype=float32>,
 'id': <tf.Tensor 'ParseSingleExample/Squeeze_id:0' shape=() dtype=int64>,
 'income': <tf.Tensor 'ParseSingleExample/Squeeze_income:0' shape=() dtype=float32>}
```

在会话执行时，为了使计算任务顺利获取到输入数据，我们需要使用 `tf.train.start_queue_runners` 方法启动执行入队操作的所有线程，具体包括将文件名入队到文件名队列的操作，以及将样例入队到样例队列的操作。这些队列操作相关的线程属于 TensorFlow 的后台线程，它们确保文件名队列和样例队列始终有数据可以供后续操作读取。下面我们补全读取 TFRecords 文件数据的代码 reader.py：

```
init_op = tf.global_variables_initializer()
sess = tf.Session()
```

```
sess.run(init_op)
# 启动执行入队操作的后台线程
tf.train.start_queue_runners(sess=sess)
# 读取第一条数据记录
for i in range(2):
  example = sess.run(features)
  print(example)
'''
输出：
{'age': 24, 'outgo': 1024.0, 'id': 1, 'income': 2048.0}
{'age': 48, 'outgo': 2048.0, 'id': 2, 'income': 4096.0}
'''
```

虽然我们用上面的代码成功读取并输出了 stat.tfrecord 文件中的数据，但是这种方法并不适用于生产环境。因为它的容错性较差，主要体现在队列操作后台线程的生命周期“无人管理”，任何线程出现异常都会导致程序崩溃。常见的异常是文件名队列或样例队列越界抛出的 `tf.errors.OutOfRangeError`。队列越界的原因通常是读取的数据记录数量超过了 `tf.train.string_input_producer` 方法中指定的数据集遍历次数。下面是常见的 `tf.errors.OutOfRangeError` 错误示例：

```
import tensorflow as tf
# 创建文件名队列 filename_queue，并制定遍历两次数据集
filename_queue = tf.train.string_input_producer(['stat.tfrecord'], num_epochs=2)
# 省略中间过程
# 遍历 5 次数据集，读取 10 条数据记录
for i in range(10):
  example = sess.run(features)
  print(example)
'''
输出：
{'age': 24, 'outgo': 1024.0, 'id': 1, 'income': 2048.0}
{'age': 48, 'outgo': 2048.0, 'id': 2, 'income': 4096.0}
{'age': 24, 'outgo': 1024.0, 'id': 1, 'income': 2048.0}
{'age': 48, 'outgo': 2048.0, 'id': 2, 'income': 4096.0}
Traceback (most recent call last):
  File "nn.py", line 28, in <module>
    example = sess.run(features)
# 省略调用栈日志
OutOfRangeError (see above for traceback): FIFOQueue '_0_input_producer' is closed and
    has insufficient elements (requested 1, current size 0)
         [[Node: ReaderReadV2 = ReaderReadV2[_device="/job:localhost/replica:0/task:0/
             cpu:0"](TFRecordReaderV2, input_producer)]]
         [[Node: ParseSingleExample/ParseExample/ParseExample/_9 = _Recv[client_terminate
             d=false, recv_device="/job:localhost/replica:0/task:0/gpu:0", send_device=
             "/job:localhost/replica:0/task:0/cpu:0", send_device_incarnation=1,
             tensor_name="edge_18_ParseSingleExample/ParseExample/ParseExample",
             tensor_type=DT_FLOAT, _device="/job:localhost/replica:0/task:0/gpu:0"]()]]
'''
```

为了处理这种异常，我们使用 `tf.train.Coordinator` 方法创建管理多线程生命周期的协调器。协调器的工作原理很简单，它监控 TensorFlow 的所有后台线程。当其中某个线程出现异常

时，它的 should_stop 成员方法返回 True，for 循环结束。然后程序执行 finally 中协调器的 request_stop 成员方法，请求所有线程安全退出。需要注意的是，当我们使用协调器管理多线程前，需要先执行 tf.local_variables_initializer 方法对其进行初始化。为此，我们使用 tf.group 方法将它和 tf.global_variables_initializer 方法聚合生成整个程序的初始化操作 init_op。使用协调器的示例如下：

```
import tensorflow as tf
# 创建文件名队列 filename_queue，并制定遍历两次数据集
filename_queue = tf.train.string_input_producer(['stat.tfrecord'], num_epochs=2)
# 省略中间过程
# 聚合两种初始化操作
init_op = tf.group(tf.global_variables_initializer(),
                   tf.local_variables_initializer())
sess.run(init_op)
# 创建协调器
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
# 打印程序的后台线程信息
print('Threads:%s' % threads)
try:
  for i in range(10):
    if not coord.should_stop():
      example = sess.run(features)
      print(example)

except tf.errors.OutOfRangeError:
  print('Catch OutOfRangeError')
finally:
  # 请求停止所有后台线程
  coord.request_stop()
  print('Finish reading')
#  等待所有后台线程安全退出
coord.join(threads)
sess.close()
'''
输出：
Threads:[<Thread(Thread-1, started daemon 140248776427264)>, <Thread(Thread-2, started
daemon 140248768034560)>]
{'age': 24, 'outgo': 1024.0, 'id': 1, 'income': 2048.0}
{'age': 48, 'outgo': 2048.0, 'id': 2, 'income': 4096.0}
{'age': 24, 'outgo': 1024.0, 'id': 1, 'income': 2048.0}
{'age': 48, 'outgo': 2048.0, 'id': 2, 'income': 4096.0}
Catch OutOfRangeError
Finish reading
'''
```

根据输出结果来看，程序启动了两个后台线程进行队列操作。在成功输出 4 条数据记录后，程序抛出 tf.errors.OutOfRangeError 异常。在发起停止所有后台线程的请求后，程序输出 Finish reading。接着协调器等待所有后台线程安全退出，最后关闭会话。

5. 扩展阅读：TFRecords 样例数据结构

介绍 TFRecords 文件格式时已经展示了创建样例的方法，如下所示：

```
example = tf.train.Example(
    features = tf.train.Features(
        feature = {
        }
    )
)
```

样例和特征的层次性源于 TensorFlow 对数据结构的巧妙定义，我们摘取 tensorflow/core/example 目录下 example.proto 和 features.proto 文件的关键部分进行说明。其中 `Example` 内部包含一个 `Features` 成员对象，如下所示：

```
// example.proto
message Example {
  Features features = 1;
};
```

而定义在 feature.proto 中的 `Features` 由形如 `<string, Feature>` 的 `feature` 字典组成。其中，`feature` 字典的值的数据结构为 `Feature`，如下所示：

```
// feature.proto
message Features {
  // 特征字典
  map<string, Feature> feature = 1;
};

message Feature {
  // 所有特征都是以下 3 种类型之一
  oneof kind {
    BytesList bytes_list = 1;
    FloatList float_list = 2;
    Int64List int64_list = 3;
  }
};
// 每个列表存储的都是可迭代的特定类型数据
message BytesList {
  repeated bytes value = 1;
}
message FloatList {
  repeated float value = 1 [packed = true];
}
message Int64List {
  repeated int64 value = 1 [packed = true];
}
```

4.1.2　创建批样例数据的方法

经过上一节的介绍，我们最后得到了许多样例，但是这些样例需要打包聚合成批数据才能供模型训练、评估和推理使用。TensorFlow 提供的 `tf.train.shuffle_batch` 方法不仅能够使用样

例创建批数据，而且能够在打包过程中打乱样例顺序，增加随机性。因此，我们认为完整的输入流水线应该还包括一个批数据队列。下面给出完整输入流水线的伪代码示例：

```
def get_my_example(filename_queue):
  reader = tf.SomeReader()
  _, value = reader.read(filename_queue)
  features = tf.decode_some(value)
  # 对样例进行预处理
  processed_example = some_processing(features)
  return processed_example

def input_pipeline(filenames, batch_size, num_epochs=None):
  # 当 num_epochs==None 时，表示文件名队列总是可用的，一直循环入队
  filename_queue = tf.train.string_input_producer(
      filenames, num_epochs=num_epochs, shuffle=True)
  example = get_my_example(filename_queue)
  # min_after_dequeue 表示从样例队列中出队的样例个数，
  # 值越大表示打乱顺序效果越好，同时意味着消耗更多内存
  min_after_dequeue = 10000
  # capacity 表示批数据队列的容量，推荐设置：
  # min_after_dequeue + (num_threads + a small safety margin) * batch_size
  capacity = min_after_dequeue + 3 * batch_size
  # 创建批样例 example_batch
  example_batch = tf.train.shuffle_batch(
      [example], batch_size=batch_size, capacity=capacity,
      min_after_dequeue=min_after_dequeue)
  return example_batch
```

`tf.train.shuffle_batch` 方法除了上面使用的参数外，常用的还有设置线程个数的 `num_threads` 参数，设置随机化种子的 `seed` 参数，以及为入队多条样例设置的 `enqueue_many` 参数。现在我们的输入流水线补齐了创建批样例数据的最后一环。图 4-2 展示了使用完整的输入流水线处理数据的工作流程。

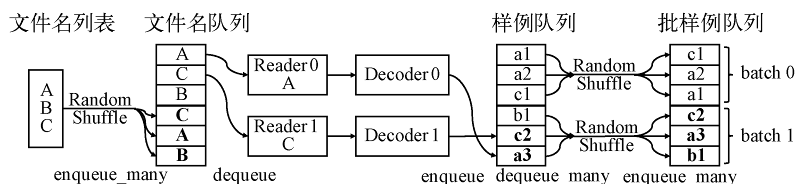


图 4-2 完整的输入流水线工作流程

4.1.3 填充数据节点的方法

使用批数据训练的模型基本都使用填充数据节点的方法，它不需要存储完整数据集，有效减少了内存开销。同时，基于输入流水线的数据读取方法保证了实时性，与将全量数据预加载到内存中并没有明显的性能差距。填充数据节点的方法很简单，只需要在会话中执行 `session.run` 方法时，向其 `feed_dict` 参数输入符合占位符操作中定义的数据即可。`feed_dict` 参数接受形

如<数据节点,填充数据>的 Python 字典。我们使用完整的输入流水线处理数据，并以填充数据节点的方式将批样例传入数据流图，流程如下所示：

```
# 使用上一节定义的 input_pipeline 方法获取批样例 x_batch
x_batch = input_pipeline(['stat.tfrecord'], batch_size=20)
# 省略创建模型步骤
sess = tf.Session()
init_op = tf.group(tf.global_variables_initializer(),
                   tf.local_variables_initializer())
sess.run(init_op)
# 创建协调器
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
try:
  for _ in range(1000):
    if not coord.should_stop():
      sess.run(train_op)
      print(example)

except tf.errors.OutOfRangeError:
  print('Catch OutOfRangeError')
finally:
  # 请求停止所有后台线程
  coord.request_stop()
  print('Finish reading')
#  等待所有后台线程安全退出
coord.join(threads)
sess.close()
```

4.1.4　处理CIFAR-10 数据集的最佳实践

本节我们以著名的图像数据集 CIFAR-10 为例，展示如何将图 4-2 中的输入流水线运用到实践中。首先，简要介绍下 CIFAR-10 数据集。它总共包含 60 000 张 32 像素×32 像素的 3 通道（RGB）彩色图片，即单张图像大小为 32×32×3=3072 字节。图片按照其主体内容一共分为 10 类，每一类 6000 张图片。这 10 类具有独特性和排他性，比如：automobile 和 truck 这两个类别没有一张重复的照片。automobile 包含轿车、跑车、SUV 等，而 truck 只包含大卡车。同时，为了避免概念混淆，两种类别都不包含小卡车或皮卡车。读者可以在 CIFAR-10 官网（http://www.cs.toronto.edu/~kriz/cifar.html）下载便于 Python、Matlab 和 C 程序处理的 3 个版本的数据集。整个数据集被分为了 6 个批数据，每一批数据都包含 10 000 张图片。其中，50 000 张用于模型训练，剩下的 10 000 张用于模型测试。

在 CIFAR-10 数据集中，一条数据记录由类别标签和图像数据两部分组成，如图 4-3 所示。我们已经知道存储单张图片需要 3072 字节，而存储类别标签需要 1 字节。因此，CIFAR-10 数据集中的单条数据记录占用 3073 字节。它们以二进制数据文件格式存储。



图 4-3 CIFAR-10 数据集和单条记录的组成结构（另见彩插）

我们使用图 4-2 中所示的输入流水线处理 CIFAR-10 数据集，程序如下所示：

```
"""处理 CIFAR-10 数据集的示例程序"""
# 4.1_best_practice.py
# -*- coding:utf-8 -*-
# 类别标签为 1 字节
LABEL_BYTES = 1
# 图片尺寸为 32×32
IMAGE_SIZE = 32
# 图片为 RGB 3 通道
IMAGE_DEPTH = 3
# 图片数据为 32×32×3＝3072 字节
IMAGE_BYTES = IMAGE_SIZE * IMAGE_SIZE * IMAGE_DEPTH
# 10 类标签
NUM_CLASSES = 10

import tensorflow as tf

def read_cifar10(data_file, batch_size):
  """从 CIFAR-10 数据文件读取批样例
  输入参数:
    data_file: CIFAR-10 数据文件
    batch_size: 批数据大小
  返回值:
    images: 形如[batch_size, image_size, image_size, 3]的图像批数据
    labels: 形如[batch_size、NUM_CLASSES]的标签批数据
  """
  # 单条数据记录大小为 1+3072=3073 字节
  record_bytes = LABEL_BYTES + IMAGE_BYTES
  # 创建文件名列表
  data_files = tf.gfile.Glob(data_file)
  # 创建文件名队列
  file_queue = tf.train.string_input_producer(data_files, shuffle=True)
  # 创建二进制文件对应的 Reader 实例，按照记录大小从文件名队列中读取样例
```

```
    reader = tf.FixedLengthRecordReader(record_bytes=record_bytes)
    _, value = reader.read(file_queue)
    # 将样例拆分为类别标签和图片
    record = tf.reshape(tf.decode_raw(value, tf.uint8), [record_bytes])
    label = tf.cast(tf.slice(record, [LABEL_BYTES]), tf.int32)
    # 将长度为 [depth * height * width] 的字符串转换为形如 [depth, height, width] 的图片张量
    depth_major = tf.reshape(tf.slice(record, [LABEL_BYTES], [IMAGE_BYTES]),
                             [IMAGE_DEPTH, IMAGE_SIZE, IMAGE_SIZE])
    # 改变图片张量各维度顺序，从 [depth, height, width] 转换为 [height, width, depth]
    image = tf.cast(tf.transpose(depth_major, [1, 2, 0]), tf.float32)
    # 创建样例队列
    example_queue = tf.RandomShuffleQueue(
        capacity=16 * batch_size,
        min_after_dequeue=8 * batch_size,
        dtypes=[tf.float32, tf.int32],
        shapes=[[IMAGE_SIZE, IMAGE_SIZE, IMAGE_DEPTH], [1]])
    num_threads = 16
    # 创建样例队列的入队操作
    example_enqueue_op = example_queue.enqueue([image, label])
    # 将定义的 16 个线程全部添加到 queue runner 中
    tf.train.add_queue_runner(tf.train.queue_runner.QueueRunner(
        example_queue, [example_enqueue_op] * num_threads))

    # 从样例队列中读取批样例图片和标签
    images, labels = example_queue.dequeue_many(batch_size)
    labels = tf.reshape(labels, [batch_size, 1])
    indices = tf.reshape(tf.range(0, batch_size, 1), [batch_size, 1])
    labels = tf.sparse_to_dense(
        tf.concat(values=[0, labels], axis=1),
        [batch_size, NUM_CLASSES], 1.0, 0.0)

    # 展示 images 和 labels 的数据结构
    assert len(images.get_shape()) == 4
    assert images.get_shape()[0] == batch_size
    assert images.get_shape()[-1] == 3
    assert len(labels.get_shape()) == 2
    assert labels.get_shape()[0] == batch_size
    assert labels.get_shape()[1] == NUM_CLASSES

    return images, labels
```

上面代码中定义的 `read_cifar10` 方法实现了从 CIFAR-10 数据集中读取批样例的功能。输入参数 `data_file` 和 `batch_size` 分别表示 CIFAR-10 数据集文件和批样例个数。返回值 `images` 和 `labels` 分别表示批样例图像数据和类别标签，如图 4-3 所示。用户可以将 `read_cifar10` 方法嵌入在自己的模型代码中，如下所示：

```
import tensorflow as tf

x = tf.placeholder(...)
y_ = tf.placeholder(...)
weight = tf.Variable(...)
bias = tf.Variable(...)
# 定义模型和优化器等
with tf.Session() as sess:
```

```
    tf.global_variables_initializer().run()
    batch_xs, batch_ys = read_cifar10(FLAGS.data_url, FLAGS.batch_size)
    sess.run(train_op, feed_dict={x: batch_xs, y_: batch_ys})
    # 打印日志
```

综上，我们基于 TensorFlow 的输入流水线技术，展示了读取 CIFAR-10 数据集的最佳实践。针对其他数据集，如 ImageNet，我们也可以使用这套并行技术提升读取数据的效率。

4.1.5 扩展阅读：MNIST数据集

由于 Python 读取数据文件的方法已经有大量资料可供参考，本节不再赘述小数据集的数据读取方法，仅给出一个典型小数据集的介绍。MNIST 是一套手写体数字的图像数据集，由纽约大学的 Yann LeCun 等人维护。该数据集中的手写体数字的大小固定为 28 像素×28 像素。图 4-4 给出了其中的典型图像示例。



图 4-4 MNIST 图像数据集中的手写体数字图像

MNIST 图像数据集使用形如 [28,28] 的二阶数组来表示每张图像，数组中的每个元素对应一个像素点。该数据集中的图像都是 256 阶灰度图，像素值 0 表示白色（背景），255 表示黑色（前景）。根据应用的需求，除了可以使用取值为 [0,255] 的 uint8 数据类型表示图像外，还可以将灰度值缩放为 [0,1] 的 float32 数据类型。以图 4-5 为例，右边的数组表示左边的手写体数字“1”的图像。

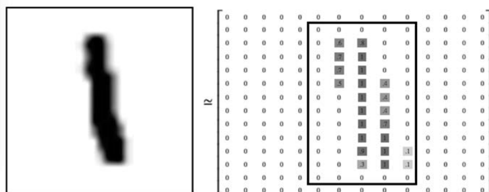


图 4-5 使用数组表示手写体数字的图像

由于每张图像的尺寸都是 28 像素×28 像素，为了方便连续存储，我们可以将形如[28,28]的二阶数组“摊平”成形如 [784] 的一阶数组。数组中的 784 个元素共同组成了一个 784 维向量。根据各分量取值的不同，这个 784 维的向量便可以表示 784×256=200 704 张不同的图像。但这些图像并非每一张都代表有效的手写体数字，其中绝大部分都是如图 4-6 所示的噪声图。



图 4-6 随机取值向量表示的噪声图

是否存在快速准确地识别手写体数字图像的算法呢？答案是肯定的。MNIST softmax 模型使用一层神经网络实现了 92% 左右的识别正确率，其优点在于计算速度快、编程难度小，适合作为机器学习的入门模型。在下一章中，我们将介绍使用 TensorFlow 实现 MNIST softmax 模型的具体流程。

4.2 模型参数

模型参数指模型的权重值和偏置值，即我们训练的对象。如果想要灵活地实现自己的训练目标，那么有必要掌握 TensorFlow 模型参数的使用方法，具体包括模型参数的创建、初始化和更新，以及从模型文件中存储和恢复模型参数的方法。其中，前 3 个方法主要由 `tf.Variable` 类实现，后两个方法由 `tf.train.Saver` 类实现。本节首先介绍模型参数的典型使用流程，然后依次介绍这 5 个使用方法，最后简要介绍变量作用域机制在处理复杂模型时的好处和方法。

4.2.1 模型参数的典型使用流程

我们将模型参数的使用流程分为如图 4-7 所示的 5 个典型步骤。图中的 `tf.Variable` 类实现了数据流图上的存储节点。它能够在操作执行完成后仍然保存变量值，故而我们使用它存储模型参数。为了训练模型，我们需要依次创建、初始化和更新模型参数。`tf.train.Saver` 是辅助训练的工具类，它实现了存储模型参数的变量与 checkpoint 文件间的读写操作。checkpoint 文件是以 <变量名，张量值> 的形式来序列化存储模型参数的二进制文件，它是用户持久化存储模型参数的推荐文件格式，扩展名为 ckpt。因此，我们一般称存储模型参数的 checkpoint 文件为模型文件。图 4-7 中的“存储”是指将变量中存储的模型参数定期写入 checkpoint 文件。通常，我们按照固定训练步数设置检查点，如每训练 10 步就将模型参数存储到 checkpoint 文件。“恢复”是指读取 checkpoint 文件中存储的模型参数，基于这些值继续训练模型。TensorFlow 提供了选择性存储和恢复部分任意变量的方法，这使得用户可以灵活地改造模型，并基于之前的训练结果做参数微调（fine-tuning）。

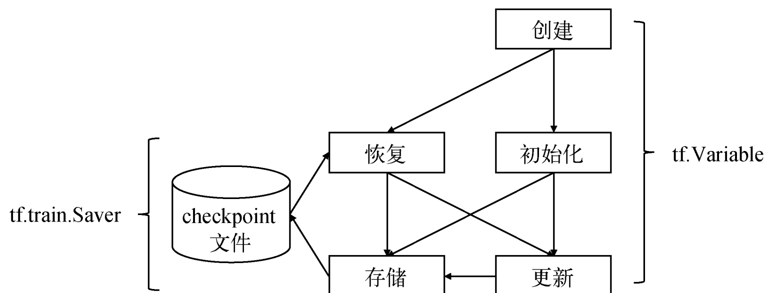


图 4-7 使用模型参数的 5 个典型步骤

4.2.2 使用tf.Variable创建、初始化和更新模型参数

为了让读者能够更深入地了解和使用 `tf.Variable` 类，本节将依次介绍使用 `tf.Variable`

类创建、初始化和更新模型参数的基本原理和具体方法。

1. 创建模型参数

创建模型参数是流程的第一步，它的主要作用是确定模型参数的基本属性，包括初始值、数据类型、张量形状、变量名称等。通常，我们使用 tf.Variable 的构造方法创建变量，并将模型参数保存在变量中。假设我们创建一个变量，它保存的权重值满足正态分布。代码如下所示：

```
import tensorflow as tf

W = tf.Variable(initial_value=tf.random_normal(shape=(1, 4), mean=100, stddev=0.35),
    name="W")
```

其中，tf.random_normal 方法返回形状为 [1,4] 的张量。它的 4 个元素符合均值为 100、标准差为 0.35 的正态分布。initial_value 参数表示在会话中为变量设置的初始值，它接受张量和生成张量的方法，如 tf.random_normal 方法；它也接受可以通过 convert_to_tensor 方法转换为张量的数据类型，如 Python 列表、元组等。除了生成符合正态分布的张量外，TensorFlow 还提供了许多生成特定张量的方法。这些张量具体包括以下 3 类：(1) 符合某种统计分布的随机张量；(2) 符合某种生成规则的序列张量；(3) 常量张量。表 4-5、表 4-6 和表 4-7 分别列出了生成这 3 类张量的方法和样例，供读者参考。

表 4-5 生成符合统计分布的随机张量方法

方 法 名	统计分布	样例代码	样例数据
tf.random_normal	正态分布	tf.random_normal([2,3],mean=-1, stddev=4)	[[0.16091979,6.71194983, -1.41309571],[-0.12917435, -2.10704994, -1.01480424]]
tf.truncated_normal	截尾正态分布	tf.truncated_normal([2,3],mean=-1, stddev=4)	[[-3.4802115,-1.4703306, -3.62793112],[-6.34977388, 2.4902215, 1.26519108]]
tf.random_uniform	均匀分布	tf.random_uniform([2,3])	[[0.6775713,0.15819943, 0.72737145],[0.22914016, 0.19896901, 0.27835727]]
tf.multinomial	多项式分布	tf.multinomial(tf.log([[10., 10.]]), 5)	[[1, 0, 0, 0, 1]]
tf.random_gamma	伽马分布	tf.random_gamma([10], [0.5, 1.5])	[[0.1702181, 0.45917341],[0.52689838, 3.80994534],[0.47716081, 1.09045565]]
tf.random_shuffle	按维度重洗	tf.random_shuffle([[1, 2], [3, 4], [5,6]])	[[3, 4],[1, 2],[5, 6]]
tf.random_crop	按形状裁剪	tf.random_crop([[1, 2], [3, 4]], [1, 2])	[[3, 4]]

表 4-6 生成符合生成规则的序列张量方法

方 法 名	初始值	生成规则	样例代码	样例数据
tf.linspace	start	(stop - start) / (num - 1)	tf.linspace(start=10.0,stop=12.0,num=3)	[10.011.0 12.0]
tf.range	start	delta	tf.range(start=3, limit=18, delta=3)	[3, 6, 9, 12, 15]

表 4-7　生成常量张量方法

方　法　名	初始值	样例代码	样例数据
tf.zeros	0	tf.zeros([3, 4], tf.int32)	[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
tf.zeros_like	0	tf.zeros_like([[1, 2, 3], [4, 5, 6]])	[[0, 0, 0], [0, 0, 0]]
tf.ones	1	tf.ones([2, 3], tf.int32)	[[1, 1, 1], [1, 1, 1]]
tf.ones_like	1	tf.ones_like([[1, 2, 3], [4, 5, 6]])	[[1, 1, 1], [1, 1, 1]]
tf.fill	value	tf.fill([2, 3], 9)	[[9, 9, 9], [9, 9, 9]]
tf.constant	value	tf.constant(-1.0, shape=[2, 3])	[[-1.0, -1.0, -1.0], [-1.0, -1.0, -1.0]]

除了直接传值和使用生成张量的方法外，我们还可以使用初始化的变量值作为新创建变量的初始值。对于已经初始化的变量，其成员方法 initialized_value 返回的变量值可以直接传入新变量的构造方法。使用 initialized_value 方法的示例如下：

```
import tensorflow as tf

W = tf.Variable(tf.random_normal(shape=(1, 4), stddev=0.35), name="W")
# 使用 W 的值初始化新变量 w_replica
w_replica = tf.Variable(W.initialized_value(), name="w_replica")
# 使用 W 的 2 倍值初始化新变量 w_twice
w_twice = tf.Variable(W.initialized_value() * 2.0, name="w_twice")
```

但是，用户不能直接使用变量实例作为新变量的初始值。因为在会话中执行初始化操作时，两个变量会出现循环依赖的问题。

当我们调用 tf.Variable 的构造方法时，程序向数据流图添加了存储节点。在上一章中简单介绍过，存储节点是可以被展开的子图，它具体包含 3 个操作和 1 个初始值。这里我们使用 TensorBoard 可视化创建权重值 W 的数据流图，如图 4-8 所示。因为我们使用 tf.random_normal 返回的随机张量初始化 W，所以它的子图不再包含初始值，而是依赖一个前置操作 random_normal。

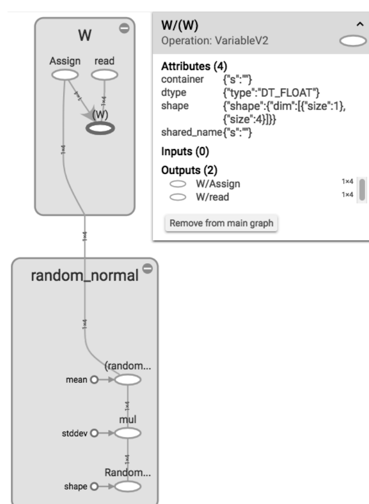


图 4-8　创建权重值 W 的数据流图

当我们在会话中执行初始化操作（如 tf.global_variables_initializer）时，程序内部调用变量 W 的 Assign 操作，而 Assign 操作又依赖 random_normal 操作。因此，初始化变量的完整过程是程序执行 random_normal 操作，将返回的张量输入到 Assign 操作，最后 Assign 操作将这些张量赋值到变量 W。下面详细解释这个执行步骤的内部实现。

2. 初始化模型参数

我们知道没有初始化的变量是无法使用的。因为创建变量仅仅定义了一个“空壳”，所以需要将数据以张量的形式传入变量。在会话维护的运行环境中完成的这个过程，我们称为初始化变量。TensorFlow 提供了两种初始化变量的选择，一种是传入初始值，然后执行初始化操作赋值，另一种是从 checkpoint 文件中恢复变量的值。让我们先把目光聚焦在前者，下一节再介绍如何从 checkpoint 文件中恢复变量。

我们知道许多关于生成初始值的方法，但是还未系统介绍初始化操作。在诸多方法中，我们最常用的初始化操作 tf.global_variables_initializers 被公认为是最简便的方法。只要在会话中执行它，程序就会初始化全局的变量。示例如下：

```
import tensorflow as tf

weights = tf.Variable(tf.random_normal(shape=(1, 4), stddev=0.35), name="weights")
biases = tf.Variable(tf.zeros([4]), name="biases")
# 创建会话
with tf.Session() as sess:
  # 初始化全局变量
  sess.run(tf.global_variables_initializer())
  print(sess.run([weights, biases]))
'''
输出：
[array([[-0.02929228, -0.37313604, -0.10337462,  0.06409851]], dtype=float32),
 array([ 0.,  0.,  0.,  0.], dtype=float32)]
'''
```

除了一次初始化所有的变量外，有时我们可能仅需要初始化部分变量。针对这种情况，我们可以调用 tf.variables_initializer 方法，并显式设置想要初始化的变量列表 var_list。如下所示：

```
import tensorflow as tf

weights = tf.Variable(tf.random_normal(shape=(1, 4), stddev=0.35), name="weights")
biases = tf.Variable(tf.zeros([4]), name="biases")
# 创建会话
with tf.Session() as sess:
  # 初始化 weights
  sess.run(tf.variables_initializer([weights]))
  print(sess.run(weights))
'''输出：[[-0.36972174 -0.85970068 -0.24391153 -0.46784061]]'''
```

变量列表 var_list 本质上是同类变量的集合。在创建变量时，我们可以通过 collections 参数显式指定变量所属的集合类别，不同类别集合拥有不同的关键字。如果没有指定 collections

参数，它的默认值为 GraphKeys.GLOBAL_VARIABLES，即将该变量添加到全局变量集合。同样，如果显式指定 trainable 参数为 True，则该变量被加入训练参数的集合，类别关键字为 GraphKeys.TRAINABLE_VARIABLES。表 4-8 列出了 TensorFlow 内置的 5 类变量集合。通过将变量归类到不同集合，用户可以快速找到具有某类相同特征的变量。比如，调用 tf.global_variables 方法，我们就能快速得到全局变量的集合。

表 4-8　TensorFlow 内置的 5 类变量集合

方法名称	类别关键字	类别说明
tf.global_variables	GraphKeys.GLOBAL_VARIABLES	跨设备的全局变量集合
tf.local_variables	GraphKeys.LOCAL_VARIABLES	进程内本地变量集合
tf.model_variables	GraphKeys.MODEL_VARIABLES	进程内存储模型参数的变量集合
tf.trainable_variables	GraphKeys.TRAINABLE_VARIABELS	存储需要训练的模型参数的变量集合
tf.moving_average_variables	GraphKeys.MOVING_AVERAGE_VARIABLES	使用指数移动平均的变量集合

基于这一认识，tf.global_variables_initializer 方法的内部实现也就不难猜到。我们在 tensorflow/python/ops/variables.py 文件中可以找到其方法定义：

```
def global_variables_initializer():
  return variables_initializer(global_variables())
```

该方法的本质是调用 tf.variables_initializer 方法，通过将 var_list 设置为 global_variables 方法返回的全局变量集合，实现了全局变量的初始化操作。类似的封装方法（如 tf.local_variables_initializer）也是同样的原理。值得一提的是，在执行分布式训练任务时，tf.global_variables_initializers 方法能够实现跨设备的全局变量初始化。

当我们需要检验变量是否成功初始化时，TensorFlow 提供了相关的判断和断言方法，如表 4-9 所示。

表 4-9　调试变量状态的方法

方法名称	功能说明
tf.is_variable_initialized	检查变量是否初始化
tf.report_uninitialized_variables	获取未初始化的变量集合
tf.assert_variables_initialized	断言变量已经初始化

3. 更新模型参数

我们知道模型参数都保存在变量中，因此更新模型参数主要指更新变量中存储的模型参数。变量是数据流图中的存储节点，也是 3 种节点中唯一有状态的节点。对于无状态的节点，如计算节点，它的输出由输入张量和节点操作共同确定。因为节点表示的操作是不变的，所以无状态节点的输出由输入张量唯一确定。比如：

```
a = tf.placeholder(tf.float32)
b = a * a
```

在不改变乘法操作的情况下，b 的值由输入张量 a 唯一确定。对于有状态的节点，如存储节点，它的输出还会受到内部保存的模型参数的影响。比如：

```
x = tf.placeholder(tf.float32)
W = tf.Variables(...)
b = tf.Variables(...)
y = matmul(W, x) + b
```

在不改变矩阵乘法和加法操作的情况下，y 的值由输入 x、变量 W 和变量 b 共同确定。即使输入 x 不变，我们也不能确定输出的 y 值。按照上面的方法计算 y 值，只是读取了变量 W 和变量 b 的值，而不会更新它们的值。

更新模型参数本质上就是对变量中保存的模型参数重新赋值。表 4-10 列出了 TensorFlow 更新变量的方法，具体包括直接赋值、加法赋值和减法赋值 3 种方法。其中，后两者本质上是对直接赋值方法的封装。

表 4-10 更新模型参数的方法

方法名称	功能说明
tf.assign	直接赋值
tf.assign_add	加法赋值
tf.assign_sub	减法赋值

下面演示了在数据流图不变的情况下，存储节点的状态不同导致相同输入得到不同输出的示例。这里使用 tf.assign_add 操作更新变量 W，代码如下所示：

```
import tensorflow as tf
# 创建变量 W
W = tf.Variable(0.0, name='W')
double = tf.multiply(2.0, W)
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  # 循环执行 4 次加法赋值操作
  for i in range(4):
    sess.run(tf.assign_add(W, 1.0))
    print('W=%s, double=%s' % (sess.run(W), sess.run(double)))
'''
输出：
W=1.0, double=2.0
W=2.0, double=4.0
W=3.0, double=6.0
W=4.0, double=8.0
'''
```

根据输出结果来看，每次循环执行的 tf.assign_add 操作都成功修改了变量 W 的值。即使数据流图和 multiply 操作输入的张量 x 固定不变，输出结果 double 也随着变量 W 的增大而改变。图 4-9 是我们使用 TensorBoard 渲染的上述操作对应的数据流图。

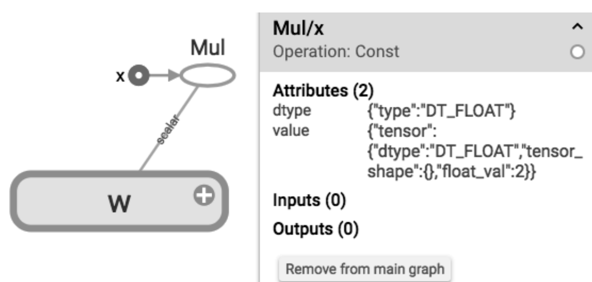


图 4-9　更新变量 W 的数据流图

在我们使用 TensorFlow 训练模型时，优化器的 `apply_gradients` 成员方法内部也会调用表 4-8 中的方法完成模型参数的更新。在多线程中训练模型时，可能涉及对变量的并发读写。考虑到这一点，TensorFlow 为变量提供了线程安全的更新方法，通过加锁机制确保数据一致性。`tf.assign` 方法的输入参数 `use_locking` 的默认值为 `True`，即 TensorFlow 默认对并发读写加锁。

4.2.3　使用tf.train.Saver保存和恢复模型参数

`tf.train.Saver` 是辅助训练的工具类，它实现了存储模型参数的变量和 checkpoint 文件间的读写操作。当我们创建 `Saver` 实例时，它的构造方法会向当前的数据流图中添加一对操作 `SaveOp` 和 `RestoreOp`。其中，`SaveOp` 负责向 checkpoint 文件中写入变量，`RestoreOp` 负责从 checkpoint 文件中读取变量。

1. 保存模型参数

`Saver` 拥有许多可配置参数，表 4-11 列出了其构造方法中的主要输入参数。在创建 `Saver` 实例时，我们可以通过 `var_list` 参数设置想要存储的变量集合，默认为全局变量集合。我们知道变量在 checkpoint 文件中是以 <变量名, 张量值> 的键值对形式序列化存储的。其中，变量名的默认值为变量操作的名称，不便于我们未来恢复时寻找特定变量。因此，我们应该为变量重命名一个有实际意义的名称。需要注意的是，`var_list` 中的变量不能出现重复名称的情况，否则会导致错误。在数据结构方面，`var_list` 支持 Python 字典和列表两种类型。

表 4-11　Saver 构造方法的主要输入参数

参数名称	功能说明	默　认　值
var_list	Saver 存储的变量集合	全局变量集合
reshape	是否允许从 checkpoint 文件中恢复时改变变量形状	True
sharded	是否将 checkpoint 文件中的变量轮循放置在所有设备上	True
max_to_keep	保留最近的检查点的个数	5
restore_sequentially	是否按顺序恢复所有变量，当模型较大时顺序恢复可以降低内存使用	True

创建 `Saver` 实例后，如果想要存储会话中当前时刻的变量值，可以调用 `Saver.save` 方法。

使用 Saver 存储变量的示例如下：

```
import tensorflow as tf
# 创建变量 W
W = tf.Variable(0.0, name='W')
double = tf.multiply(2.0, W)
# 创建 Saver
saver = tf.train.Saver({'weights':W})
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  for i in range(4):
    sess.run(tf.assign_add(W, 1.0))
    # 存储变量 W
    saver.save(sess, '/tmp/summary/test.ckpt')
```

我们把每一次循环更新后的变量 W 都存储到 test.ckpt 文件中。下面介绍如何从 checkpoint 文件中恢复变量 W。

2. 恢复模型参数

当需要基于某个 checkpoint 文件继续训练模型时，应该使用 Saver.restore 方法恢复文件中的变量值，而不是使用 initializer 方法为它们设置初始值。Saver.restore 方法也需要显式设置加载变量值的会话和存储变量的 checkpoint 文件路径。使用 Saver 恢复 test.ckpt 文件中变量 W 的示例如下：

```
import tensorflow as tf
# 创建变量 W
W = tf.Variable(0.0, name='weights')
double = tf.multiply(2.0, W)
# 创建 Saver
saver = tf.train.Saver()
with tf.Session() as sess:
  # 恢复变量 W 的值
  saver.restore(sess, '/tmp/summary/test.ckpt')
  print('restored:W=%s' % sess.run(W))
  for i in range(4):
    sess.run(tf.assign_add(W, 1.0))
    print('W=%s, double=%s' % (sess.run(W), sess.run(double)))
'''
输出：
restored:W=4.0
W=5.0, double=10.0
W=6.0, double=12.0
W=7.0, double=14.0
W=8.0, double=16.0
'''
```

test.ckpt 文件中最新检查点存储的变量 W 的值为 4.0。根据输出结果来看，恢复操作执行成功。另外，因为我们已将变量 W 重命名为 weights，所以恢复时必须将变量 W 的名称设置为 weights，否则会导致错误。

4.2.4　使用变量作用域处理复杂模型

我们已经学会使用 `tf.Variable` 类创建和使用变量，并掌握了模型参数的典型使用流程，这足以应付许多常见的模型训练场景。但是，当你需要编写深度神经网络模型时，这种瀑布流式的模型定义方法就无法轻松应对了。因此，本节将介绍如何使用更灵活和具有层次化结构的变量作用域来帮助我们处理更加复杂的模型。

1. tf.Variable 的局限

下面以图像识别的 CNN 网络为例说明这个问题。为简单起见，假设模型仅包含两层卷积层。模型代码如下所示：

```
def my_image_filter(input_images):
  # 第 1 层卷积
  conv1_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]),
      name="conv1_weights")
  conv1_biases = tf.Variable(tf.zeros([32]), name="conv1_biases")
  conv1 = tf.nn.conv2d(input_images, conv1_weights,
      strides=[1, 1, 1, 1], padding='SAME')
  relu1 = tf.nn.relu(conv1 + conv1_biases)
  # 第 2 层卷积
  conv2_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]),
      name="conv2_weights")
  conv2_biases = tf.Variable(tf.zeros([32]), name="conv2_biases")
  conv2 = tf.nn.conv2d(relu1, conv2_weights,
      strides=[1, 1, 1, 1], padding='SAME')
  return tf.nn.relu(conv2 + conv2_biases)
```

即使是两层的神经网络，模型仍然包含 4 个不同的模型参数，分别是 `conv1_weights`、`conv1_biases`、`conv2_weights` 和 `conv2_biases`。我们的代码复杂度将随着网络层数不断增加。如果需要定义 100 层的深度神经网络，这种方法的问题就更加明显。

同时，这种方法还存在模型复用的问题。假设想要多次使用该模型，`tf.Variable` 方法在每次调用模型时都会创建一份变量，但它们存储的是相同的模型参数。如下所示：

```
# 第一次调用创建 4 个变量
result1 = my_image_filter(image1)
# 第二次调用再次创建 4 个变量
result2 = my_image_filter(image2)
```

随着模型复用次数的增加，内存开销也将不断上升，直到内存溢出。一种简单的解决方法是定义一个存储所有模型参数的 Python 字典 `variables_dict`，然后在每次调用时都使用 `variables_dict` 中的共享参数。如下所示：

```
variables_dict = {
    "conv1_weights": tf.Variable(tf.random_normal([5, 5, 32, 32]),
        name="conv1_weights")
    "conv2_weights": tf.Variable(tf.random_normal([5, 5, 32, 32]),
        name="conv2_weights")
    "conv1_biases": tf.Variable(tf.zeros([32]), name="conv1_biases")
```

```
    "conv2_biases": tf.Variable(tf.zeros([32]), name="conv2_biases")
}

def my_image_filter(input_images, variables_dict):
  conv1 = tf.nn.conv2d(input_images, variables_dict["conv1_weights"],
      strides=[1, 1, 1, 1], padding='SAME')
  relu1 = tf.nn.relu(conv1 + variables_dict["conv1_biases"])

  conv2 = tf.nn.conv2d(relu1, variables_dict["conv2_weights"],
      strides=[1, 1, 1, 1], padding='SAME')
  return tf.nn.relu(conv2 + variables_dict["conv2_biases"])

# 两次调用都使用 variables_dict 中的模型参数
result1 = my_image_filter(image1, variables_dict)
result2 = my_image_filter(image2, variables_dict)
```


虽然这种方法看似解决了模型复用的问题，但是它破坏了模型的封装性，降低了代码的可读性。一旦网络结构发生变化，我们需要分别修改模型方法 my_image_filter 和参数字典 variables_dict，以及调用模型的代码。同时，该方法也没有解决网络层数增加带来的代码复杂度上升的问题。

2. 变量作用域的好处

事实上，网络层数即使增加到 1000 层，常用的网络类型也只有 10 种左右。一个很自然的想法是编写管理各类网络的方法，在这个方法内部定义该类网络的结构和参数。同时，该方法还应该在复用模型时，允许共享该层的模型参数。TensorFlow 的变量作用域机制以一种优雅的、轻量级的、非侵入式的方式实现了上述想法，有效地解决了 tf.Variable 方法不能应付的两个问题。TensorFlow 的变量作用域机制主要由 tf.get_variable 方法和 tf.variable_scope 方法实现。前者负责创建或获取指定名称的变量，后者负责管理传入 tf.get_variable 方法的变量名称的名字空间。

tf.get_variable 方法的主要输入参数是 name、shape 和 initializer，分别表示变量的名称、形状和初始化方法。区别在于 tf.Variable 方法直接使用初始值 initial_value，tf.get_variable 方法在运行时根据张量的形状动态初始化变量。常用的初始化方法如 tf.constant_initializer、tf.random_uniform_initializer 和 tf.random_normal_initializer，它们分别表示使用常量值、区间值和符合正态分布的张量初始化变量。我们使用 tf.get_variable 方法定义卷积层方法，如下所示：

```
def conv_relu(input, kernel_shape, bias_shape):
  # 创建或获取名叫 weights 的变量
  weights = tf.get_variable("weights", kernel_shape,
      initializer=tf.random_normal_initializer())
  # 创建或获取名叫 biases 的变量
  biases = tf.get_variable("biases", bias_shape,
      initializer=tf.constant_initializer(0.0))
  conv = tf.nn.conv2d(input, weights,
      strides=[1, 1, 1, 1], padding='SAME')
  return tf.nn.relu(conv + biases)
```

3. 变量作用域的使用

现在，我们可以使用 conv_relu 方法创建两层卷积的图像识别模型，如下所示：

```
def my_image_filter(input_images):
  with tf.variable_scope("conv1"):
    # 创建 conv1/weights 和 conv1/biases 变量
    relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
  with tf.variable_scope("conv2"):
    # 创建 conv2/weights 和 conv2/biases 变量
    return conv_relu(relu1, [5, 5, 32, 32], [32])
```

with 上下文中定义的变量都会加上 tf.variable_scope 方法中定义的前缀，这样能够通过不同的变量作用域区分同类网络的不同层参数。但是，这还不能解决模型复用的问题。如果第二次调用 my_image_filter 方法，tf.get_variable 方法就会抛出变量已存在，无法再定义的错误。如下所示：

```
ValueError: Variable conv1/weights already exists, disallowed. Did you mean to set reuse=True
    in VarScope?
```

因此，我们应该显式设置 tf.variable_scope 方法的 reuse 参数为 True，表示共享该作用域内的参数。当第二次调用 tf.get_variable 方法时，能够正确返回共享参数。如下所示：

```
def my_image_filter(input_images):
  # 共享第 1 层卷积的所有参数
  with tf.variable_scope("conv1", reuse=True):
    relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
  # 共享第 2 层卷积的所有参数
  with tf.variable_scope("conv2", reuse=True):
    return conv_relu(relu1, [5, 5, 32, 32], [32])
```

变量作用域也支持嵌套定义，如下所示：

```
with tf.variable_scope("foo"):
  with tf.variable_scope("bar"):
    v = tf.get_variable("v", [1])
assert v.name == "foo/bar/v:0"
```

除了使用 tf.variable_scope 方法共享作用域下的模型参数外，我们还可以为作用域内的所有变量设置初始化方法，如下所示：

```
with tf.variable_scope("foo", initializer=tf.constant_initializer(0.4)):
  v = tf.get_variable("v", [1])
  # 使用外层定义的初始化方法
  assert v.eval() == 0.4
  W = tf.get_variable("W", [1], initializer=tf.constant_initializer(0.3))
  # tf.get_variable 显式设置的初始化方法覆盖作用域定义的方法
  assert W.eval() == 0.3
  with tf.variable_scope("bar"):
    v = tf.get_variable("v", [1])
    # 继承外层作用域的初始化方法
    assert v.eval() == 0.4
  with tf.variable_scope("baz", initializer=tf.constant_initializer(0.2)):
    v = tf.get_variable("v", [1])
    # 使用优先级更高的内层定义的初始化方法
    assert v.eval() == 0.2
```

在处理结构复杂的模型时，恰当地使用变量作用域可以简化模型的定义和初始化工作。同时，这也有助于开发更加层次化和模块化的模型，提升代码可读性。

4.3　命令行参数

本节所述的命令行参数特指启动 TensorFlow 程序时输入的参数。按照功能不同，可以将其分为模型超参数和集群参数两种。前者指机器学习和深度学习模型中的框架参数，比如梯度下降的学习率和批数据大小等，主要用于优化模型的训练精度和速度。后者指运行 TensorFlow 分布式任务的集群配置参数，如参数服务器主机地址和工作服务器主机地址等，主要用于设置 TensorFlow 集群。

4

本节介绍解析命令行参数的两种主流解决方案：argparse 和 tf.app.flags。前者 是 Python 标准库中功能强大的命令行参数解析模块，也是广大 Python 用户的优先选择。后者是 TensorFlow 基于 argparse 封装的工具类，特点是简单易用，新手也能快速掌握。

4.3.1　使用argparse解析命令行参数

argparse 模块是 Python 标准库提供的用于命令行参数与选项解析的模块。它默认会从 sys.argv 中解析出我们在程序中事先定义的参数，并触发对应的动作（如保存参数、统计参数次数和回调函数等）。相比用户手动编程实现命令行参数的解析，使用 argparse 模块具有以下 4 个优点：

- 同时支持定位参数和关键字参数的解析；
- 自动生成使用方法信息；
- 自动生成帮助信息；
- 自动抛出错误提示信息。

为了更优雅地开发基于 Python API 的 TensorFlow 应用程序，读者有必要学会使用 argparse 模块解析命令行参数。

argparse 模块的核心是参数解析器（ArgumentParser），简称解析器。用户事先需要向解析器中添加想要解析的参数。在程序启动时，解析器就能“按图索骥”地从命令行中解析出对应的参数。下面我们详细介绍使用 argparse 模块的方法。该模块的典型使用流程分为以下 3 步：

(1) 创建解析器；

(2) 添加待解析参数；

(3) 解析命令行输入的参数。

下面简要介绍这 3 个步骤并给出具体的案例。

1. 创建解析器

首先，需要创建 argparse 模块中的解析器实例。它的构造方法的主要输入参数如表 4-12 所

示，该方法接受多个输入参数，用于设置程序帮助文本的描述信息。其中，usage 参数是程序的使用方法，description 参数是程序的功能说明。

表 4-12　解析器构造方法的主要输入参数

参数名称	功能说明
prog	程序的文件名
usage	程序使用方法，默认根据用户添加的参数自动生成
description	程序功能说明
epilog	程序描述末尾的补充信息
add_help	是否为程序添加用于打印帮助信息的-h 和--help 参数
argument_default	所有参数的全局默认值

为了直观地展示表 4-12 中各输入参数的实际效果，我们在 Python 的交互式编程环境中介绍它们的使用方法和对应输出。首先，创建一个解析器，同时设置相关的输入参数。下面的代码展示了表 4-12 中各输入参数打印时的真实效果：

```
import argparse

parser = argparse.ArgumentParser(prog='demo', description='A demo program',
    epilog='The end of usage')
parser.print_help()      # 打印程序帮助信息
"""输出
usage: demo [-h]         # 程序名称由 prog 参数定义

A demo program           # 使用方法由 description 参数定义

optional arguments:      # 默认为程序添加-h 和--help 参数
  -h, --help  show this help message and exit

The end of usage         # 补充信息由 epilog 参数定义
"""
```

因为我们没有显式设置 usage 参数，所以程序的使用方法直接基于待解析的参数生成。又因为没有向解析器中添加参数，所以帮助信息中只有 -h 和 --help 参数的使用说明。下面我们介绍如何向解析器中添加待解析的参数。

2. 添加待解析参数

定义待解析参数，并将其正确添加到解析器是最重要的一步。每调用一次解析器的 add_argument 方法，我们就向解析器中新添加了一个参数。argparse 同时支持定位参数和关键字参数的解析。成功解析命令行参数时，可以触发不同的动作，具体动作由 add_argument 方法的 action 参数指定。

add_argument 方法的原型如下所示：

```
ArgumentParser.add_argument(name or flags...[, action][, nargs][, const][, default][, type]
    [, choices][, required][, help][, metavar][, dest])
```

表 4-13 列出了该方法的输入参数及其功能说明。

表 4-13 add_argument 方法的输入参数

参数名称	功能说明
name or flags	名称或标记
action	解析参数成功时触发的动作
nargs	待解析参数的个数
const	action 和 nargs 参数可能使用的常量值
default	待解析参数的默认值
type	解析参数后保存的数据类型
choices	参数可选值集合，用于约束枚举型参数
required	是否为必须从输入中解析的参数，默认为 False
help	参数功能说明
dest	解析参数后保存的对象名称

这里重点分析最常用的两个参数 name or flags 和 action。

- name or flags：表示定位参数的名称或关键字参数的标记，它是一个可变长的字符串列表(长度至少为 1)。对于定位参数，命令行参数的顺序与添加参数的顺序必须保持一致。对于关键字参数，没有顺序要求，只需要正确提供关键字参数的标记（如 -t、--type）即可。它们类似于 Python 方法中经常使用的默认参数（*args）和可变参数（**kwargs)。
- action：表示解析参数成功时触发的动作，具体包括以下 7 种。
  - store：保存参数值（默认动作）。
  - store_const：保存 const 参数设置的值。
  - store_true/store_false：保存布尔值（True/False）。
  - append：将参数值追加到列表。
  - append_const：将 const 参数设置的值追加到列表。
  - count：为关键字参数计数。
  - version：打印程序的版本信息。

需要特别说明的是：程序启动时，成功解析参数的前提是已向解析器添加过参数。但是，已添加到解析器的参数未必都在程序启动时输入参数值,这是因为我们可以使用 default 为参数设置默认值。只要添加参数时没有设置 required=True，就可以自由选择是否在程序启动时输入该参数。

现在，我们使用 add_argument 方法向第一步中定义的解析器添加待解析参数，代码如下所示：

```
parser.add_argument('name')
parser.add_argument('-a', '--age', type=int, required=True)
```

```
parser.add_argument('-s', '--status', choices=['alpha', 'beta', 'released'], type=str,
dest='myStatus')
parser.add_argument('-v', '--version', action='version', version='%(prog)s 1.0')
parser.print_help()
'''输出
usage: demo [-h] -a AGE [-s {alpha,beta,released}] [-v] name

positional arguments:
  name                     # 定位参数

optional arguments:        # 关键字参数
  -h, --help            show this help message and exit
  -a AGE, --age AGE        # 必须输入的参数
  -s {alpha,beta,released}, --status {alpha,beta,released}
  -v, --version         show program's version number and exit

The end of usage
'''
```

根据帮助信息中的程序使用方法，启动程序时必须输入的参数是 name 和 -a（或 --age)。[] 框内表示可选参数，即使用户启动程序时不输入，程序也只是设置其为 None 而不会报错。

3. 解析参数

解析参数就是按照解析器中的参数名称或标记，将参数字符串先分类再匹配的过程。当匹配成功时，解析器会将参数转换为当前名字空间中的对象。解析器的 parse_args 方法实现了解析参数字符串的功能。当我们调用 parse_args 方法时，解析器按照 add_argument 方法中定义的参数属性（即表 4-11 中的 type 和 dest 字段）创建对象。对于名字空间，读者可以将其理解为一个 Python 字典。字典中存储的是形如<对象名，参数值>的键值对。

结合前面步骤，现在我们能够定义一个完整的参数解析程序 demo.py，如下所示：

```
"""demo.py"""
# -*- coding: utf-8 -*-
import argparse

parser = argparse.ArgumentParser(prog='demo', description='A demo program', epilog='The end
of usage')

parser.add_argument('name')
parser.add_argument('-a', '--age', type=int, required=True)
parser.add_argument('-s', '--status', choices=['alpha', 'beta', 'released'], type=str,
dest='myStatus')

args = parser.parse_args() # 将名字空间赋值给 args
print(args) # 输出名字空间
```

parse_args 方法支持解析多种格式的参数字符串。对于定位参数，该方法并无格式要求。对于关键字参数，除了第二步中展示的以空格分隔标记和参数值（如 -a AGE）的格式外，parse_args 方法还支持以等号分隔（如 -a=AGE）的格式和不分隔（如 -aAGE）的格式。

现在，可以在启动 demo.py 程序时输入几组不同的参数字符串，查看它们返回的名字空间和

解析结果。如下所示：

```
$ python demo.py Jony -a 23
Namespace(age=23, myStatus=None, name='Jony')
$ python demo.py Alice --age 10
Namespace(age=10, myStatus=None, name='Alice')
$ python demo.py Alice --age 10 -sbeta
Namespace(age=10, myStatus='beta', name='Alice')
```

同时，parse_args 方法也会对参数字符串做输入检查，包括参数数据类型是否有效、关键字参数的标记是否正确等。一旦发现输入的参数字符串有误，解析器立刻抛出错误信息并显示正确的使用方法。示例如下所示：


```
# 无效数据类型
$ python demo.py Alice -a 10.0
usage: demo [-h] -a AGE [-s {alpha,beta,released}] [-v] name
demo: error: argument -a/--age: invalid int value: '10.0'
# 无效标记
$ python demo.py Alice -a 10 --location hangzhou
usage: demo [-h] -a AGE [-s {alpha,beta,released}] [-v] name
demo: error: unrecognized arguments: --location hangzhou
# 参数个数错误
$ python demo.py
usage: demo [-h] -a AGE [-s {alpha,beta,released}] [-v] name
demo: error: too few arguments
```

在某些情况下，用户输入的命令行参数可能是给级联调用的多个程序使用的。为此，解析器实现了只解析预定义参数字符串的 parse_known_args 方法，它会将解析器中未添加的参数保存在一个字符串数组中。该方法同时返回一个名字空间和一个字符串数组，后者可以直接传输给调用链中的后续程序来解析和使用。

我们将 demo.py 中的 parse_args 方法替换为 parse_known_args 方法，并保存为 new_demo.py 文件，代码如下所示：

```
"""new_demo.py"""
# ……创建解析器，添加参数……
args, unparsed = parser.parse_known_args() # 将解析器中未定义的参数返回给 unparsed
print('args=%s, unparsed=%s' % (args, unparsed))
```

现在，输入解析器中未定义的参数 location 和对应值 hangzhou，此时程序的输出结果如下所示：

```
$ python new_demo.py Alice -a 10 --location hangzhou
args=Namespace(age=10, myStatus=None, name='Alice'), unparsed=['--location', 'hangzhou']
```

4. 最佳实践

下面我们以 MNIST softmax 模型为例，展示如何在 TensorFlow 程序中使用 argparse 模块解析命令行参数。这里使用 parse_known_args 方法将解析成功的参数命名空间赋值给 FLAGS。命令行输入的数据目录参数解析后，就会保存在 FLAGS.data_dir 对象中；未解析的参数保存在 unparsed 字符串数组中。下面省略了创建模型和训练模型等中间过程，仅展示命令行参数解

析相关的代码：

```
"""argparse_mnist.py"""
import argparse

from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf

FLAGS = None

def main(_):
  # 导入数据
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
  # 省略中间步骤

if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir', type=str,
                      default='/tmp/tensorflow/mnist/input_data',
                      help='Directory for storing input data')
  FLAGS, unparsed = parser.parse_known_args()
```

4.3.2　使用tf.app.flags解析命令行参数

虽然 argparse 模块功能强大，但是它对于很多 Python 新手来说可能比较复杂。在机器学习和深度学习场景下，一般的程序并不需要如此灵活而完备的命令行参数解析机制。为此，TensorFlow 封装了一套基于 argparse 的参数解析模块——tf.app.flags（以下简称 flags）。该模块简化了 argparse 中解析器的大量配置选项，仅实现参数解析、默认值和打印帮助信息等基本功能。如果说 argparse 是灵活多样可配置，那么 flags 则是简单直接易上手。

使用 flags 解析参数不需要创建解析器，然后添加参数，再调用解析参数的方法。相反，用户仅需调用 flags 模块中定义参数的方法即可，其他工作由 flags 模块内部完成。下面我们介绍使用 flags 解析参数的流程和方法。

首先，将 tf.app.flags 模块的引用赋值到 flags，以简化编码。接着，调用 flags 模块中定义字符串参数的方法 DEFINE_string。它的 3 个输入参数分别表示参数名称、默认值和使用方法。在程序启动时，如果我们定义的参数解析成功，那么它们都会保存在 flags.FLAGS 名字空间中。为了进一步简化代码形式，我们为 flags.FLAGS 取一个别名，如 FLAGS。如果在程序启动时输入了正确的参数字符串，就可以成功访问 FLAGS 名字空间中的参数对象。下面的代码展示了这个流程：

```
"""flags_demo.py"""
import tensorflow as tf

flags = tf.app.flags
flags.DEFINE_string("data_dir", "/tmp/mnist-data",
                    "Directory for storing mnist data")
FLAGS = flags.FLAGS # 将解析成功的参数保存到 flags.FLAGS 名字空间中
def main(_):
```

```
  print(FLAGS.data_dir)
if __name__ == "__main__":
  tf.app.run()
```

现在，我们在命令行中测试 flags 解析参数的实际效果。命令及输出如下所示：

```
$ python flags_demo.py
/tmp/mnist-data
$ python flags_demo.py --data_dir /data/mnist
/data/mnist
# flags 为程序自动生成使用方法和帮助信息
$ python flags_demo.py -h
usage: flags_demo.py [-h] [--data_dir DATA_DIR]

optional arguments:
  -h, --help           show this help message and exit
  --data_dir DATA_DIR  Direcotry for storing mnist data
```


表 4-14 列出了 flags 模块定义参数的方法。其中，DEFINE_bool 方法是 DEFINE_boolean 方法的别名，可以将其视为同一个方法。这些方法的输入参数一致，均为 flags_name、default_value 和 docstring，分别表示参数名称、默认值和使用说明。这 4 种方法的内部实现也十分相似，它们均调用了 flags 模块内的 _define_helper 方法。

表 4-14 flags 模块定义参数的方法

方法名称	调用样例
DEFINE_float	flags.DEFINE_float("learning_rate", 0.01, "Learning rate")
DEFINE_integer	flags.DEFINE_integer("batch_size", 100, "Training batch size")
DEFINE_string	flags.DEFINE_string("data_dir", "/tmp/mnist-data","Directory for storing data")
DEFINE_boolean/DEFINE_bool	flags.DEFINE_boolean("inference_only", False, "Only perform inferencing"

下面我们以定义字符串参数的 DEFINE_string 方法为例，说明这类方法的实现原理：

```
import argparse as _argparse
# 创建全局解析器
_global_parser = _argparse.ArgumentParser()
# 定义字符串参数方法，该方法内部调用了 _define_helper 方法
def DEFINE_string(flag_name, default_value, docstring):
  _define_helper(flag_name, default_value, docstring, str)
# _define_helper 方法是对解析器 add_argument 方法的封装
def _define_helper(flag_name, default_value, docstring, flagtype):
  _global_parser.add_argument('--' + flag_name,
                              default=default_value,
                              help=docstring,
                              type=flagtype)
```

从 _define_helper 方法的实现来看，它内部调用 add_argument 方法添加关键字参数。并且，它仅设置 4 个输入参数：flags、default、help 和 type。其中，前 3 个参数由用户定义，type 则由 DEFINE_* 方法自动填充。使用 flags 模块解析命令行参数时，不需要显式调用 parse_args 或 parse_unknown_args 方法解析参数，因为 tf.app.run 方法内部会调用 flags.FLAGS._parse_flags 方法解析命令行参数。

下面我们解释 flags.FLAGS 的实现原理。它是 _FlagValues 类的实例，该类内部创建了保存参数对象的字典。利用 Python 的反射机制，程序内部将解析成功的所有参数都设置为 _FlagValues 实例的属性。感兴趣的读者可以查看 tensorflow/python/platform/flags.py 文件，了解 flags 模块的具体实现。

当我们启动一个 TensorFlow 程序并向其传入命令行参数时,程序内部首先创建 _FlagValues 实例，接着调用 _parse_flags 成员方法解析命令行参数。程序每解析一个 DEFINE_* 方法定义的参数，立刻将其加入 _FlagValues 实例内部的参数字典。最终，当我们使用 FLAGS 实例的属性（如 FLAGS.data_dir）时，程序内部调用内建方法 __getattr__ 返回对应的参数对象。相关代码如下：

```
class _FlagValues(object):
  """全局关键字参数的寄存器和访问器"""
  def __init__(self):
    # 存储参数对象的字典
    self.__dict__['__flags'] = {}
    # 判断是否解析过命令行参数的标志
    self.__dict__['__parsed'] = False

  def _parse_flags(self, args=None):
    """解析用户添加的关键字参数，剩余的参数直接传递给其他程序"""
    result, unparsed = _global_parser.parse_known_args(args=args)
    for flag_name, val in vars(result).items():
      self.__dict__['__flags'][flag_name] = val
    self.__dict__['__parsed'] = True
    return unparsed

  def __getattr__(self, name):
    """根据'--name'关键字取得参数值"""
    if not self.__dict__['__parsed']:
      self._parse_flags()
    if name not in self.__dict__['__flags']:
      raise AttributeError(name)
    return self.__dict__['__flags'][name]

  def __setattr__(self, name, value):
    """设置'--name'关键字的参数值"""
    if not self.__dict__['__parsed']:
      self._parse_flags()
    self.__dict__['__flags'][name] = value

# 访问全局关键字参数的对象 FLAGS
FLAGS = _FlagValues()
```

接着，我们使用 flags 模块重写上一节的最佳实践，以便对比两种方法的代码量和配置选项差异：

```
"""flags_mnist.py"""
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
```

```
flags = tf.app.flags
flags.DEFINE_string("data_dir", "/tmp/mnist-data",
                    "Directory for storing mnist data")
FLAGS = flags.FLAGS
def main(_):
  # 导入数据
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
  # 省略中间步骤

if __name__ == "__main__":
  tf.app.run()
```

对比可以发现，使用 `flags` 模块只需在文件开头部分定义待解析参数的名称、默认值和使用方法。调用解析成功的参数也很简单，直接访问保存在 `FLAGS` 名字空间中的对象即可。因此，对于没有太多个性化需求和多样化配置的情况，使用 `FLAGS` 模块解析命令行参数更加简单明了。

4.4 小结

针对输入数据集、模型参数和命令行参数这三类不同的数据，TensorFlow 提供了简单易用的数据处理模块。基于 `Reader` 和 `Queue` 的输入流水线及多线程并发机制提升了大数据集的处理效率，基于 `Constant` 或 `Variable` 抽象的变量预加载则可以降低小数据集计算时的访存开销。模型参数的创建、初始化和更新由 `tf.Variable` 类实现,模型参数的存储和恢复由 `tf.train.Saver` 类实现，模型参数一般存储于 checkpoint 文件中。命令行参数的解析有两种解决方案：`argparse` 和 tf.app.flags。Python 标准库的 `argparse` 模块灵活性好，功能强大，广受软件开发者喜爱；TensorFlow 内置的 `tf.app.flags` 模块则更加容易上手，适合算法研究人员使用。

第 5 章

TensorFlow 编程框架

经过前面的学习，我们已经认识了 TensorFlow 的基础抽象和对应的数据流图元素，并初步掌握了 TensorFlow 的数据处理方式和简单模型开发方法。读者心中的 TensorFlow 知识脉络想必已经逐步形成。在此基础上，本章介绍 TensorFlow 的编程框架，以一种更加整体而系统的视角串联前述概述，帮助读者理解 TensorFlow 应用层 API 及上层运行时组件的工作原理，从而实现正确使用框架与工具、熟练开发应用程序的目标。依据程序运行时形态的差异，本章将分述 TensorFlow 单机程序和分布式程序的编程框架，以两类编程框架的关键工作步骤为主线，深入剖析其设计思想，重点讲解数据并行、同步和异步训练，以及长周期训练管理等机制。

5.1 单机程序编程框架

本节中，我们就来介绍一下单机程序的编程框架。这里从代码开发视角切入，将 TensorFlow 数据流图中的各种元素和概念映射到一个典型的单机程序——MNIST softmax 模型的训练与推理程序。

5.1.1 概述

单机程序是指启动和运行都仅在一台机器的一个进程中完成的程序。这种程序的模型参数只在本地 CPU/GPU 的内存之间传输，没有网络通信的开销，非常适合参数不多、计算量小的模型。相比分布式程序，单机程序的设计更加简单，编程难度更低，有利于新手快速入门。

图 5-1 展示了使用 TensorFlow 单机程序编程框架的关键步骤。以开发流程中操作的核心对象为依据，编写单机程序主要分为创建单机数据流图（`Graph`）和创建单机会话（`DirectSession`）两步。

- **创建单机数据流图（模型）**。数据流图主要由 3 类节点组成，分别是表示输入数据集的占位符（`placeholder`）、保存模型参数的变量（`Variable`），以及前向图和后向图中的计算操作（`Operation`）。
- **创建并运行单机会话**。会话中执行的操作具体包括变量的初始化操作（`init_op`）、模型的恢复和保存操作、单步推理（`inference_op`）和单步训练（`train_op`）操作等。

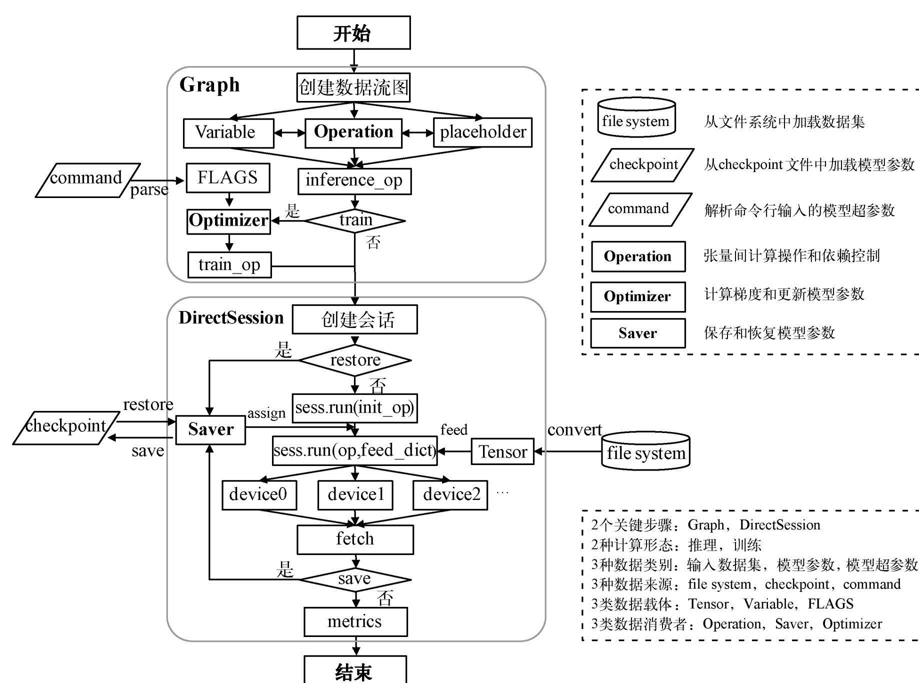


图 5-1 使用单机程序编程框架的关键步骤

单机程序的计算形态主要分为推理（预测）态和训练态。前者仅执行推理操作，计算得到推理或预测结果；后者依次执行前向图的推理计算和后向图的梯度计算，并完成模型参数的更新。

TensorFlow 单机程序处理的数据主要分为 3 类，分别是来自于文件系统的输入数据集、从 checkpoint 文件恢复的模型参数和从命令行解析的模型超参数。在数据流图中，它们分别使用张量、变量和 `FLAGS` 名字空间来保存。在会话中，它们作为输入数据分别传输给操作、`Saver` 和优化器这 3 类数据消费者。其中，操作定义了张量间的运算，如矩阵相乘、激活函数、神经网络层等。操作之间有可能存在依赖控制，用于管理多个操作的逻辑先后关系，确保以正确的时序执行各个操作。`Saver` 负责管理 checkpoint 文件。它能够从 checkpoint 文件中恢复模型参数到变量，或者将变量中的模型参数保存到 checkpoint 文件。优化器负责计算梯度和更新模型参数。常见的优化器有 SGD、Adam、Adagrad、Adadelta 等。表 5-1 对比了 TensorFlow 单机程序处理的 3 类数据的类别、来源、载体和消费者间的映射关系。

表 5-1 TensorFlow 单机程序处理的 3 类数据

数据类别	数据来源	数据载体	数据消费者
输入数据集	文件系统	张量	操作
模型参数	checkpoint 文件	变量	`Saver`
模型超参数	命令行	`FLAGS` 名字空间	优化器

5.1.2　创建单机数据流图

对于深度学习和机器学习程序来说，模型是程序的主体。模型本身由数据流图中的节点组成，节点具体包括表示输入数据集的占位符、保存模型参数的变量，以及定义前向图和后向图的计算操作等。创建数据流图是使用 TensorFlow 单机程序编程框架的第一个关键步骤。下面我们介绍一下具体步骤和方法。

TensorFlow 内部会为当前程序的上下文维护一个默认的数据流图实例。同时，它也允许用户在程序中显式创建多个数据流图实例。如果希望显式创建数据流图，并指定特定的数据流图为当前上下文的默认数据流图，那么可以使用以下代码实现：

```
# 创建数据流图 g1
g1 = tf.Graph()
with g1.as_default():
  # 在该上下文中，默认使用数据流图 g1
  a = tf.Variable(0, name='a')
  # 断言变量 a 被添加到数据流图 g1 中
  assert a.graph is g1
# 创建数据流图 g2
with tf.Graph().as_default() as g2:
  # 在该上下文中，默认使用数据流图 g2
  b = tf.Variable(0, name='b')
  # 断言变量 b 被添加到数据流图 g2 中
  assert b.graph is g2
```

这里 `tf.Graph` 方法返回新建的数据流图实例，`as_default` 成员方法将自身设置为当前上下文的默认数据流图。用户可以使用 `with` 语句管理不同数据流图实例的上下文环境。所有操作实例都实现了 `graph` 成员方法，用于获取当前操作所属的数据流图。因此，我们可以使用断言语句判断变量 `a` 和 `b` 是否被分别添加到了数据流图 `g1` 和 `g2` 中。

如果用户没有显式创建数据流图实例，那么程序使用的将是 TensorFlow 内部创建的默认数据流图实例。在这种情况下，用户定义的所有操作都会被添加到默认实例中。下面展示的 MNIST softmax 模型的数据流图创建代码即为这种情况：

```
"""5.1_best_practice.py"""
# -*- coding: utf-8 -*-
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf

flags = tf.app.flags
flags.DEFINE_string("data_dir", "/tmp/mnist-data",
                    "Directory for storing mnist data")
flags.DEFINE_float("learning_rate", 0.5, "Learning rate")
FLAGS = flags.FLAGS

def main(_):
  # 创建 MNIST 数据集实例
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
  # 创建模型
```

```
x = tf.placeholder(tf.float32, [None, 784])    # 图像数据
W = tf.Variable(tf.zeros([784, 10]))           # 模型权重
b = tf.Variable(tf.zeros([10]))                # 模型偏置
y = tf.matmul(x, W) + b                        # 推理操作
y_ = tf.placeholder(tf.float32, [None, 10])    # 图像标签
# 使用交叉熵作为损失值
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
# 创建梯度下降优化器
optimizer = tf.train.GradientDescentOptimizer(FLAGS.learning_rate)
# 定义单步训练操作
train_op = optimizer.minimize(cross_entropy)
```

这段代码的工作流程简述如下。

首先，从命令行解析 MNIST 数据目录参数，并将其保存在 FLAGS 名字空间中。后续代码可以通过 FLAGS.data_dir 属性获取这一参数。

然后，从数据目录中读取 MNIST 数据集。深度学习和机器学习算法使用的数据集通常可以根据用途划分为训练集、验证集和测试集。在 TensorFlow 附带的数据集处理代码中，MNIST 数据集的分类和保存分别由 tensorflow/contrib/learn/python/learn/datasets/mnist.py 文件定义的 read_data_sets 方法和 DataSet 类实现。其中，read_data_sets 方法负责解析 MNIST 数据集，并将其划分为训练集、验证集和测试集，分别保存到各自的 DataSet 实例中。DataSet 类提供 next_batch 成员方法。当用户在会话中向占位符填充数据时，可以通过这一方法从 DataSet 实例中获取批数据。为了统一管理 MNIST 的 3 个数据集，tensorflow/contrib/learn/python/learn/datasets/base.py 文件利用 namedtuple 定义了 Datasets 元组：Datasets = collections.namedtuple('Datasets', ['train', 'validation', 'test'])。本例中，read_data_sets 方法返回一个名为 mnist 的 Datasets 元组，用户可以调用 mnist.train 成员访问 MNIST 训练集。

接着，我们依次创建手写体数字图像的占位符、保存模型权重和偏置的变量、前向图的推理操作、手写体数字标签的占位符、用作损失值的交叉熵，以及梯度下降优化器。其中，手写体数字标签是长度为 10 的 one-hot 向量，它的 10 个分量分别对应 0~9 这 10 个数字。10 个分量中只有一个值为 1，其余值为 0。值为 1 的分量的下标即是对应的手写体数字。例如，[0,0,0,0,0,0,0,0,0,1] 表示该数字为 9。

最后，将极小化损失值的计算操作定义为单步训练操作。训练操作汇聚了模型单步训练的所有操作，我们通常将其命名为 train_op 或 train_step。上述步骤中的各个操作都是训练操作的前置依赖。如果我们已经得到一个训练好的模型，就不需要定义交叉熵和优化器，直接执行推理操作即可。推理操作汇聚了模型单步推理的所有操作，我们通常将其命名为 inference_op 或 inference_step。因为推理操作的输出张量正好是模型输出，所以也可以将其直接命名为 y。需要注意的是，上面代码创建的推理操作 y = matmul(x,W)+b 并不是 MNIST softmax 模型的最终输出，MNIST softmax 模型的完整的推理操作应该是 y = softmax(matmul(x,W)+b)。为了提升计算效率，我们可以使用 tf.nn.softmax_cross_entropy_with_logits 操作，将 softmax 函数

合并到交叉熵的归约计算中，代码如下所示：

```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
```

如果不希望合并 softmax 函数，也可以分别定义推理操作和交叉熵，相关代码如下：

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y)),
    reduction_indices=[1]))
```

现在我们已经学会了创建数据流图的方法。下面我们将介绍如何在会话中执行训练和推理操作。

5.1.3　创建并运行单机会话

这是使用 TensorFlow 单机程序编程框架的第二个关键步骤，涉及的操作包括变量初始化操作、模型的恢复和保存操作、单步推理和单步训练操作等。为了执行数据流图中的操作，首先需要创建单机会话实例。

根据运行时形态的不同，会话分为单机会话（DirectSession）和分布式会话（GrpcSeesion），两者都是基于 BaseSession 类实现的。在 BaseSession 类的构造方法中，创建会话实例是通过调用 C API 定义的 TF_NewSession 方法实现的。下面是 tensorflow/python/client/session.py 文件定义的 BaseSession 类构造方法的代码，我们摘取出创建会话的关键部分：

```
from tensorflow.python import pywrap_tensorflow as tf_session

class BaseSession(SessionInterface):
  def __init__(self, target='', graph=None, config=None):
    # 这里省略了数据流图加载和会话配置过程
    self._session = None
    opts = tf_session.TF_NewSessionOptions(target=self._target, config=config)
    try:
      with errors.raise_exception_on_not_ok_status() as status:
        # 使用 C API 中定义的 TF_NewSession 方法创建实例
        self._session = tf_session.TF_NewSession(opts, status)
    finally:
      tf_session.TF_DeleteSessionOptions(opts)
```

在创建会话时，C API 定义的统一入口方法不能感知上层 Python API 中会话类的类型。为了区分单机会话和分布式会话，BaseSession 类的构造方法需要将输入参数 target 传入 C API 的 TF_NewSession 方法。如果 target 为 None（默认值）或空字符串，则创建单机会话；如果 target 被赋予 gRPC 服务的 URL，则创建分布式会话。因此，之前创建单机会话时，不对 target 参数进行赋值。

下面我们以将上一节介绍的 MNIST softmax 模型为例，展示单机会话的创建流程。首先，将 MNIST softmax 模型的数据流图加载到新建的单机会话中。接着，初始化全局变量，设置最大训练步数为 1000 步。在每一步迭代训练时，都从预处理的 MNIST 数据集中取出一批手写体图像数

据 batch_xs 和标签数据 batch_ys，并分别填充到对应的占位符 x 和 y_ 中。下面的代码展示了在会话中不断执行单步训练操作的详细过程：

```
# 创建 Saver
saver = tf.train.Saver()
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
# 最大训练步数
for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
  # 每 100 步保存一次模型参数
  if i % 100 = 0:
    saver.save(sess, 'mnist.ckpt')
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print('acc=%s' % sess.run(accuracy,
                          feed_dict={x: mnist.test.images,
                                     y_: mnist.test.labels}))
```

当我们调用 sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys}) 语句执行单步训练操作时，程序内部首先提取单步训练操作依赖的所有前置操作。这些操作的节点共同组成一幅子图。然后，程序会将子图中的计算节点、存储节点和数据节点按照各自的执行设备分类，相同设备上的节点组成了一幅局部图。每个设备上的局部图在实际执行时，根据节点间的依赖关系将各个节点有序地加载到设备上执行。对于单机程序来说，相同机器上不同编号的 CPU 或 GPU 就是不同的设备，我们可以在创建节点时指定执行该节点的设备。下面的代码展示了如何使用 tf.device 方法指定节点的执行设备：

```
# 在 0 号 CPU 上执行的存储节点
with tf.device("/cpu:0"):
  v = tf.Variable(...)
# 在 0 号 GPU 上执行的计算节点
with tf.device("/gpu:0"):
  z = tf.matmul(x, y)
```

同时，tf.device 方法也支持嵌套使用。如果我们希望把计算量小的存储节点和数据节点放在 CPU 上执行，把计算量大的计算节点放在 GPU 上执行，那么可以参考下面的代码实现：

```
with tf.device("/cpu:0"):
  x = tf.placeholder(...)
  w = tf.Variable(...)
  b = tf.Variable(...)
  with tf.device("/gpu:0"):
    y = tf.matmul(w,x) + b
  y_ = tf.placeholder(...)
  ...
```

本例中，我们没有使用之前保存的模型参数继续训练，而是直接调用 tf.global_variables_initilizer 方法初始化所有的模型参数。如果想要从 checkpoint 文件中恢复模型参数继续训练，还需要在训练开始前调用 Saver 对象的 restore 成员方法恢复模型参数值。为此添加的代码如

下所示：

```
saver.restore(sess, 'mnist.ckpt')
for i in range(1000):
  # 训练模型
```

在 restore 方法内部，我们调用了变量的 assign 方法，继而更新了变量中的模型参数值。随着迭代训练步数的增加，损失值不断减小，模型在训练集上预测的准确率也会逐渐上升。训练结束后，我们需要评估模型是否在训练数据集上过拟合。于是，我们新建正确预测个数（correct_prediction）和准确率（accuracy）这两个操作，然后计算模型在测试数据集上的真实值和推理值的差异，以便考查模型在测试数据集上的准确率，检验模型的泛化能力。程序运行后，得到下面展示的输出结果：

```
Extracting /tmp/mnist-data/train-images-idx3-ubyte.gz
Extracting /tmp/mnist-data/train-labels-idx1-ubyte.gz
Extracting /tmp/mnist-data/t10k-images-idx3-ubyte.gz
Extracting /tmp/mnist-data/t10k-labels-idx1-ubyte.gz
acc=0.9161
```

可以看出，MNIST softmax 模型在测试集上获得了 91.61% 的准确率。

通过编程实践，我们深入理解了 TensorFlow 单机程序开发流程中的两个核心对象——数据流图与会话。前者是算法模型的载体，后者为模型提供了运行环境。如果读者想要独立实现单机运行的深度学习和机器学习模型，那么可以参考本节的编程框架进行设计和开发。

5.2　分布式程序编程框架

TensorFlow 的细粒度 API 为分布式程序的设计与开发提供了高度的灵活性，用户可以将不同的分布式架构应用于 TensorFlow 程序。PS-worker 是一种经典的分布式架构，它在大规模分布式机器学习、深度学习系统领域得到了广泛的应用。TensorFlow 提供对 PS-worker 架构支持，并将其作为其推荐的、标准的分布式编程框架。

本节主要介绍 TensorFlow 分布式程序的编程框架，以流程图的形式展现分布式程序的关键步骤。目前，大多数分布式程序均采用数据并行的模式加速模型训练，TensorFlow 也为此专门设计了具有高可用性的同步优化器，以便用户快速构建分布式程序。TensorFlow 提供的 Supervisor 以灵活易用的方式提升了模型训练过程的健壮性，同时简化了模型微调和重训练的流程。本节理论与实践相结合，由浅入深地引导读者掌握 TensorFlow 分布式程序编程框架的设计思想、工作流程和使用方法。

5.2.1　PS-worker架构概述

PS-worker 架构有效解决了大规模参数在分布式存储和更新时的一致性问题，兼具通用性和高效性。典型的 PS-worker 架构如图 5-2 所示，所有模型参数唯一地存储在 PS 的内存中。当模型参数规模超过一台服务器的内存大小时，则需要分布式地存储在多个 PS 中。最简单的分布式存

储策略为循环（round-robin）：按照用户定义模型参数的顺序，将参数依次循环保存到各 PS 中，尽可能保证每个 PS 存储的参数个数相等。假如模型一共有 9 个参数，按照循环策略分布式存储在 3 个 PS 中，则各 PS 存储的参数分别是 [1, 4, 7]、[2, 5, 8] 和 [3, 6, 9]。模型训练过程中的主要计算——前向图推理计算和后向图梯度计算均由 worker 完成。训练数据一般存储在共享文件系统或分布式文件系统中（注：本节仅讨论数据并行训练模式，即各 worker 数据流图的拓扑结构相同，填充的训练数据不同）。

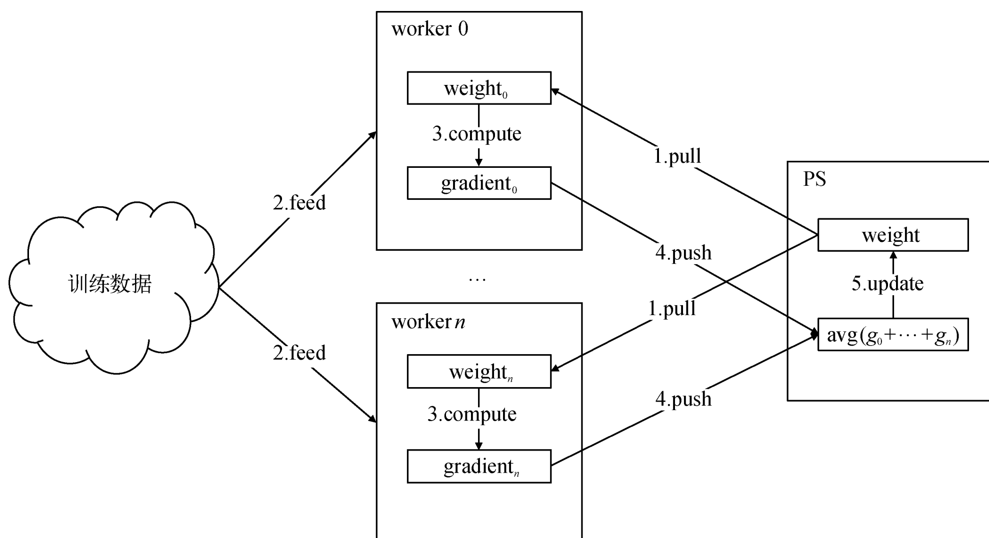


图 5-2　典型 PS-worker 架构训练分布式模型的流程

为简单起见，假设一个 PS 能够存储所有模型参数。在这种情况下，使用 PS-worker 架构训练分布式模型的流程如下。

(1) pull：各 worker 根据数据流图的拓扑结构，从 PS 拉取最新的模型参数。

(2) feed：各 worker 按照一定的规则填充不同批次的批数据。

(3) compute：各 worker 使用相同的模型参数和不同的批数据计算梯度，得出不同的梯度值。

(4) push：各 worker 将第 (3) 步中计算得到的梯度值推送到 PS。

(5) update：PS 汇总来自 n 个 worker 的总计 n 份梯度值，求出梯度平均值后更新模型参数。

分布式模型的单步训练主要由这 5 步构成。训练分布式模型的过程就是不断循环执行这 5 步的过程，直到达到最大训练步数或损失值小于阈值。

PS-worker 架构的核心思想是将模型和训练解耦合。它将一个模型的训练过程分为两类作业（job）：一类是模型相关的作业，包括模型参数的存储、分发、汇总、更新，由 PS 执行；另一类是训练相关的作业，包括推理计算和梯度计算等，由 worker 执行。同类作业内部可以包含多个

并行执行的任务（task）。每个任务的工作流程相同，但处理的数据不同。

5.2.2　分布式程序编程框架概述

分布式程序是指由多个进程协同执行的程序。在 TensorFlow 的分布式编程框架中，每个 PS 和 worker 都由单独的进程实现。图 5-3 展示了使用 TensorFlow 分布式程序编程框架的关键步骤。以开发流程中操作的核心对象为依据，编写分布式程序主要分为创建集群（`Cluster`）、创建分布式数据流图（`Graph`），以及创建并运行分布式会话（`GrpcSession`）这 3 个步骤。

- **创建集群**。包括定义集群中所有进程对应的服务器主机名（或 IP 地址）和端口，并启动各个服务。
- **创建分布式数据流图**。数据流图上的节点包括表示输入数据集的占位符、保存模型参数的变量、前向图和后向图的计算操作，以及专门为分布式模型设计的同步优化器和同步标记队列（`sync_token_queue`）的初始化操作（`sync_init_op`）。
- **创建并运行分布式会话**。涉及使用模型训练管理组件（`Supervisor`）创建分布式会话，然后在会话中执行数据流图的各项操作（如同步标记队列的初始化操作、全局变量的初始化操作、模型的恢复和保存操作、单步推理和单步训练操作等）。

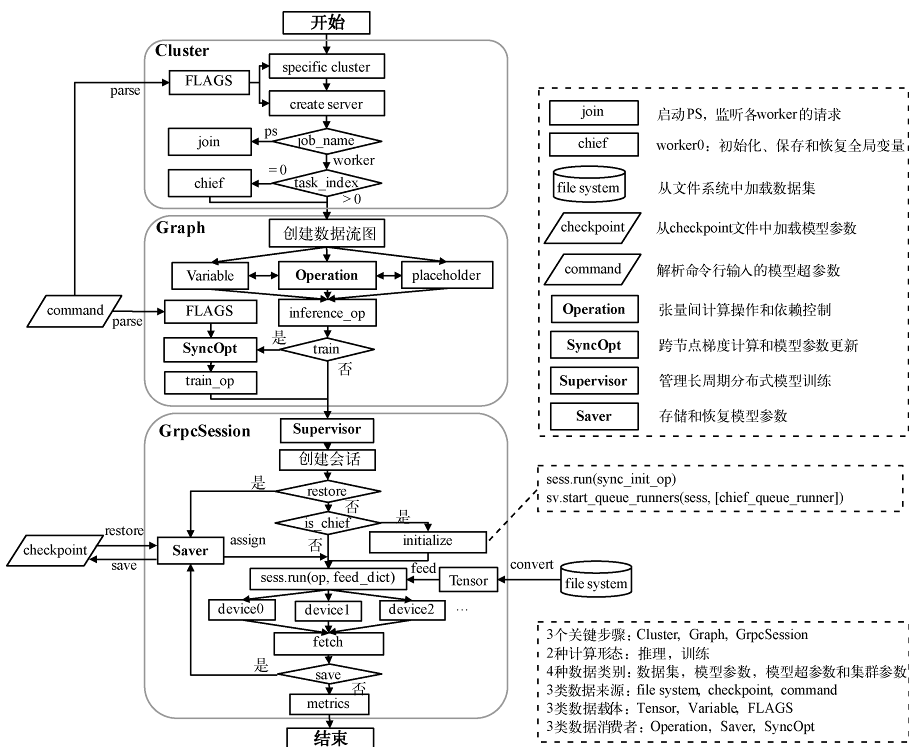


图 5-3　使用 TensorFlow 分布式程序编程框架的关键步骤

TensorFlow 分布式程序处理的数据与单机程序类似，数据来源也仍然是文件系统、checkpoint 文件和命令行参数。不同之处在于，分布式程序从命令行解析的参数除了模型超参数外，还新增了集群配置参数。TensorFlow 分布式程序使用的同步优化器是对普通优化器的封装。同步优化器的梯度计算部分由其封装的优化器实现，多 worker 间操作的依赖控制和模型参数更新模式由同步优化器的 `apply_gradients` 方法实现。下面我们逐一介绍使用 TensorFlow 分布式程序编程框架的流程，并分析其中的设计思想。

5.2.3　创建TensorFlow集群

从应用视角看，TensorFlow 集群是指完成一次完整的深度学习训练或推理工作所需的分布式运行实体及其相关的资源集合。与很多深度学习系统类似，TensorFlow 集群中的每个任务在运行时映射到操作系统中的一个进程。任务按照所属作业类型不同，可分为 PS 和 worker 两类，同类任务通过不同的任务编号（`task_index`）加以区分。通常，我们将任务编号为 0 的 worker 称为 chief worker。它与其他 worker 任务略有不同，除了按部就班地训练模型外，还需要在训练开始前初始化全局变量，以及在训练过程中将全局变量保存到 checkpoint 文件。现在可以根据作业名称（`job_name`）和任务编号唯一确定集群中的任意一个服务器。此处的“服务器”是集群中特定任务（PS 或 worker）中的一种上层抽象，而不是指物理服务器。为了方便说明，后面以 PS 任务或 worker 任务代指 TensorFlow 集群中的这种服务器抽象。

典型的分布式应用程序的进程包含两个部分。

- **服务端**：响应集群内其他任务的服务请求，负责协调会话的运行及本地设备上的局部图执行。
- **客户端**：维护用户创建的数据流图和会话，通过访问服务端执行数据流图。

TensorFlow 服务端和客户端通常并存于同一个程序中。客户端组件一般是指用户使用上层 API（如 Python、C++、Java 等）编写的模型，而服务端组件一方面包括用户编写的会话运行入口代码，同时也包括 TensorFlow 核心库内部实现的会话和数据流图执行逻辑。PS 任务通常仅执行 TensorFlow 服务端组件，而 worker 任务同时执行 TensorFlow 服务端和客户端组件。

TensorFlow 集群的逻辑部署结构如图 5-4 所示。TensorFlow 集群可以使用 `tf.train.ClusterSpec` 类进行定义，该类的构造方法接受一个描述集群中所有任务的字典，其键为作业名称，值为属于该作业类型的所有任务的主机名和端口。用户需要显式指定该字典，以确保每个任务能够以点对点方式与集群中任意其他的任务通信。TensorFlow 任务内部的服务器抽象是 `tf.train.Server` 类，其构造方法接受的主要输入参数为 `tf.train.ClusterSpec` 实例，以及当前任务的作业名称和任务编号。

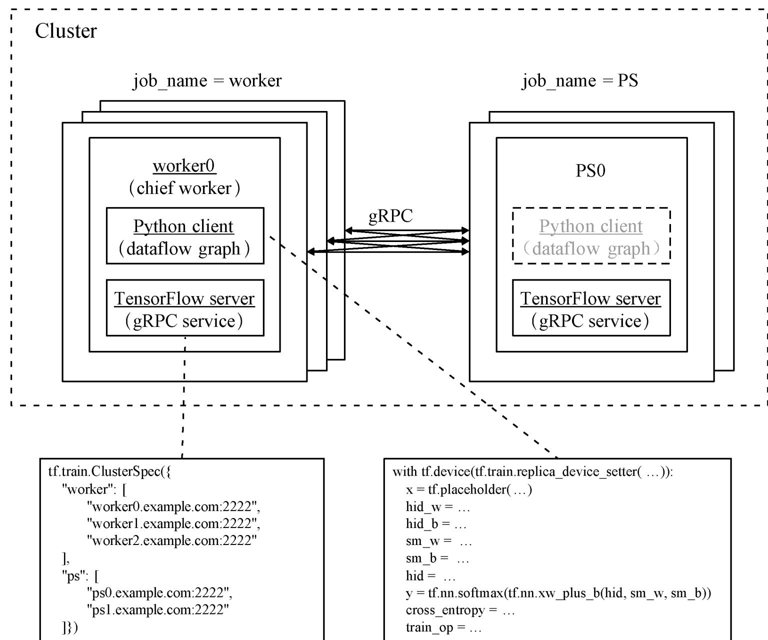


图 5-4　TensorFlow 集群的逻辑部署结构（以 Python API 为例）

因为 TensorFlow 没有提供一次性启动整个集群的解决方案，所以用户需要在每台机器上逐个手动启动一个集群的所有 PS 和 worker 任务。为了能够以同一份代码启动不同的任务，我们需要将所有 worker 任务的主机名和端口、所有 PS 任务主机名和端口、当前任务的作业名称以及任务编号这 4 个集群配置项参数化。通过输入不同的命令行参数组合，用户就可以使用同一份代码正确地启动每一个任务。典型的配置参数如表 5-2 所示。

表 5-2　TensorFlow 集群的 4 个典型配置参数

参数名称	功能说明	输入样例
job_name	作业名称	"worker"
task_index	任务编号	0
ps_hosts	所有 PS 任务的主机名和端口	"10.0.0.1:2222"
worker_hosts	所有 worker 任务的主机名和端口	"10.0.0.2:2223,10.0.0.3:2223"

现在，我们参考图 5-3 中创建集群的流程，使用表 5-2 推荐的参数名称和配置来创建 TensorFlow 集群。这里不妨将下面的文件命名为 trainer.py，这段代码的主要功能是定义并解析集群参数，从而创建并启动 TensorFlow 集群：

```
"""trainer.py"""
from tensorflow import flags
import tensorflow as tf
```

```
# 定义 TensorFlow 集群参数
flags.DEFINE_integer("task_index", None,
                     "Worker task index, should be >= 0. task_index=0 is "
                     "the master worker task the performs the variable "
                     "initialization.")
flags.DEFINE_string("ps_hosts", None,
                    "Comma-separated list of hostname:port pairs")
flags.DEFINE_string("worker_hosts", None,
                    "Comma-separated list of hostname:port pairs")
flags.DEFINE_string("job_name", None, "job name: worker or PS")
def main(unused_argv):
  # 解析集群参数 ps_hosts 和 worker_hosts
  PS_spec = FLAGS.ps_hosts.split(",")
  worker_spec = FLAGS.worker_hosts.split(",")
  # 定义 TensorFlow 集群
  cluster = tf.train.ClusterSpec({
      "PS": PS_spec,
      "worker": worker_spec})

  server = tf.train.Server(
      cluster, job_name=FLAGS.job_name, task_index=FLAGS.task_index)
  # 如果是 PS 任务，则开始监听各 worker 的请求（join 函数持续等待，不会返回）
  if FLAGS.job_name == "PS":
    server.join()
  # 如果是 worker 任务，则将任务编号为 0 的 worker 设置为 chief worker
  is_chief = (FLAGS.task_index == 0)
```

现在，如果要启动一个包含两个 PS 和两个 worker 的 TensorFlow 集群，那么可以使用如下所示的 4 条命令：

```
# On ps0.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=PS --task_index=0
# On ps1.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=PS --task_index=1
# On worker0.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=worker --task_index=0
# On worker1.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=worker --task_index=1
```

这里我们通过命令行参数指定不同进程的作业名称和任务编号。

5.2.4 将操作放置到目标设备

如果所有任务都使用同一份代码启动，那么必须将各任务的操作放置到对应的设备上执行。我们已经知道，可以使用 with tf.device 语句来指定某一代码作用域对应的目标设备。tf.device 方法的一种简单用法是直接传入代表设备名称的字符串，该字符串由作业名称、任务编号和可选的设备编号组成。典型代码如下所示：

```
# 放置变量 weights_1 和 biases_1 到 PS 0 任务所在的设备执行
with tf.device("/job:PS/task:0"):
  weights_1 = tf.Variable(...)
  biases_1 = tf.Variable(...)
# 放置变量 weights_2 和 biases_2 到 PS 1 任务所在的设备执行
with tf.device("/job:PS/task:1"):
  weights_2 = tf.Variable(...)
  biases_2 = tf.Variable(...)
# 放置操作 input、layer_1 和 logits 到 worker 2 所在的设备执行
with tf.device("/job:worker/task:2"):
  input = tf.placeholder
  layer_1 = tf.nn.relu(tf.matmul(input, weights_1) + biases_1)
  logits = tf.nn.relu(tf.matmul(layer_1, weights_2) + biases_2)
```

此外，tf.device 方法亦接受以“设备名称生成函数”作为参数。这种使用方式有利于设计高级的操作放置策略，简化复杂策略的代码实现。TensorFlow 内置了一种名为 tf.train.replica_device_setter 的设备设置器方法，它能够返回 tf.device 所接受的设备名称生成函数，主要用于在数据并行的场景下放置操作。它的主要输入参数是 worker_device、ps_device 和 cluster，分别表示 worker 任务绑定的设备名称、PS 任务绑定的设备名称和 TensorFlow 集群实例。这种方式的典型代码如下所示：

```
if FLAGS.num_gpus > 0:
  if FLAGS.num_gpus < num_workers:
    raise ValueError("number of gpus is less than number of workers")
  gpu = (FLAGS.task_index % FLAGS.num_gpus)
  worker_device = "/job:worker/task:%d/gpu:%d" % (FLAGS.task_index, gpu)
elif FLAGS.num_gpus == 0:
  cpu = 0
  worker_device = "/job:worker/task:%d/cpu:%d" % (FLAGS.task_index, cpu)

with tf.device(
    tf.train.replica_device_setter(
        worker_device=worker_device,
        ps_device="/job:ps/cpu:0",
        cluster=cluster)):
```

5.2.5 数据并行模式

目前，业界通常采用数据并行的方式加速模型训练。在这种方式下，所有 worker 共享 PS 上存储的模型参数，并按照相同拓扑结构的数据流图进行计算。因为不同的 worker 填充了不同的批数据，所以每个 worker 计算出的梯度值不一样。PS 汇总各 worker 的梯度值，然后用梯度平均

值更新模型参数。这种方式的好处是有效避免了单批数据的噪音对最优化方向的影响，相当于同时派出多个人参考不同批次数据，寻找梯度下降最快的方向，并综合所有梯度，找出令所有人满意的最优化方向。

根据数据流图的构建模式，我们可以将数据并行分为以下两类。

- **图内复制（in-graph replication）**：单进程、“单机多卡”的数据并行训练，需要用户自己实现梯度的汇总和平均计算，典型实例如 models/tutorials/image/cifar10/cifar10_multi_gpu_train.py 中 CIFAR-10 的多 GPU 训练。
- **图间复制（between-graph replication）**：多进程、可跨多机的分布式并行训练，用户需要调用同步优化器（`SyncReplicasOptimizer`）实现分布式模型的梯度计算和模型参数更新，典型实例如 tensorflow/tools/dist_test/python/mnist_replica.py 中的 MNIST 分布式模型（以下简称 MNIST replica 模型）。

根据训练时的模型参数更新机制，我们可以将数据并行分为以下两类。

- **异步训练（asynchronous training）**：每个 worker 独立训练，计算出梯度值后立即（或按照预置的策略等待特定事件发生后）进行模型参数更新计算。每个 worker 无需阻塞等待其他所有 worker 的梯度计算完成。
- **同步训练（synchronous training）**：每个 worker 独立训练，直到所有 worker 计算出梯度值后再进行模型参数的汇总计算，并更新当前训练步的模型参数。计算较快的 worker 需要阻塞等待计算较慢的 worker。

两种训练机制都同时适用于图内复制和图间复制模式。通常情况下，同步训练比异步训练的收敛速度更快，训练步数更少；异步训练的单步耗时更少，但容易受到单批数据的影响，训练步数反而更多。

5.2.6 同步训练机制

我们已经学习了 TensorFlow 单机模型的训练方法。在此基础上，本节主要介绍分布式模型的同步训练机制。同步训练机制基于同步标记队列和同步优化器，它们能够确保分布式模型每一步的训练过程都是公平和公正的，不受个别计算能力强的 worker 任务的影响。

我们一般将决定是否使用同步训练的超参数命名为 `FLAGS.sync_replicas`。如果它等于 `True`，则表示使用同步训练机制更新参数。这种情况下，同步优化器自动将各 worker 任务计算出的梯度值进行汇总，然后使用梯度的平均值进行模型参数的更新。否则，使用异步训练机制，这时所有 worker 独自训练，仅共享模型参数。下面的代码展示了如何使用同步优化器实现 MNIST 模型的同步训练：

```
# 创建数据流图，得出推理结果 y
y = tf.nn.softmax(tf.nn.xw_plus_b(hid, sm_w, sm_b))
# 使用交叉熵评估两个概率分布间的相似性
```

```
# 因为概率取值范围为 [0, 1]，为避免出现无意义的 log(0)，故将 y 值裁剪到 [1e-10, 1.0]
cross_entropy = -tf.reduce_sum(y_ * tf.log(tf.clip_by_value(y, 1e-10, 1.0)))
# 使用 Adam 做最优化求解
opt = tf.train.AdamOptimizer(FLAGS.learning_rate)
# 如果使用同步训练机制，则创建同步优化器
if FLAGS.sync_replicas:
  if FLAGS.replicas_to_aggregate is None:
    replicas_to_aggregate = num_workers
  else:
    replicas_to_aggregate = FLAGS.replicas_to_aggregate
  # 创建名为 mnist_sync_replicas 的同步优化器
  opt = tf.train.SyncReplicasOptimizer(
      opt,
      replicas_to_aggregate=replicas_to_aggregate,
      total_num_replicas=num_workers,
      name="mnist_sync_replicas")
# 如果使用异步训练机制，则各 worker 直接使用 Adam 优化器独自训练，仅共享模型参数
train_step = opt.minimize(cross_entropy, global_step=global_step)
```

如上面的代码所示，用户需要在已定义的普通优化器的基础上，再创建一个包含普通优化器和各项配置参数的同步优化器实例。表 5-3 列出了同步优化器的主要配置参数。用户通常需要从命令行输入这些配置参数，然后利用 4.3.2 节介绍的 flags 模块对其进行解析。

表 5-3　同步优化器的主要配置参数

参数名称	功能说明	默　认　值
replicas_to_aggregate	并行副本数	num_workers
total_num_replicas	实际副本数或 worker 任务数	num_workers
nonevariable_averages	是否使用 ExponentialMovingAverage	None
variables_to_average	需要进行平均值计算的模型参数列表	None

表 5-3 中的 `num_workers`（worker 任务数）是用户在集群配置的 `worker_hosts` 参数中声明的 worker 任务数量，可通过解析命令行得到。TensorFlow 的 worker 任务是承担梯度计算的实体，副本是模型训练过程中单独处理一份批数据的抽象。下面我们介绍表 5-3 中两个极易混淆的参数：并行副本数和实际副本数。

- **并行副本数**：单步训练中需要参与并行计算的副本数量，即用户期望单步训练的并行数据个数。
- **实际副本数**：单步训练中实际参与计算的 worker 任务数。

通过配置不同的并行副本数和实际副本数，TensorFlow 为用户提供了灵活的同步训练机制及多样化的模型参数更新模式。典型的模式包括以下 3 种。

- **`replicas_to_aggregate=total_num_replicas`**：“全民参与”模式。每个 worker 任务都领取一份不同的批数据进行训练，每一步训练一般都会由所有 worker 共同参与计算。
- **`replicas_to_aggregate>total_num_replicas`**：“能者多劳”模式。因为并行副本数大于 worker 任务数，所以每一步训练都存在计算能力强的 worker 计算多份梯度的情况。在

每一步训练中，计算能力强的worker将自己本次计算得出的梯度推送到PS后，如果发现尚未训练的批数据，就会再次领取一份批数据进行训练。当PS接收的梯度值份数达到并行副本数后，它会计算得出梯度的平均值，然后更新模型参数，进入下一步训练。

- **replicas_to_aggregate<total_num_replicas**：“替补等位”模式，因为并行副本数小于worker任务数，所以个别worker不需要参与训练。在某些worker任务出现异常时，空闲的worker可以作为“替补”，确保TensorFlow分布式训练过程的高可用。

同步优化器在计算梯度时与普通优化器没有区别，均直接调用基类 `Optimizer` 的 `compute_gradients` 成员方法。但是，在应用梯度更新模型参数时，它为分布式程序做了特别的设计。同步优化器重写了 `Optimizer` 的 `apply_gradients` 成员方法，感兴趣的读者可以阅读tensorflow/python/training/sync_replicas_optimizer.py文件对该方法的定义。图5-5展示了同步优化器中最常用的参数更新模式——“全民参与”模式的工作流程。

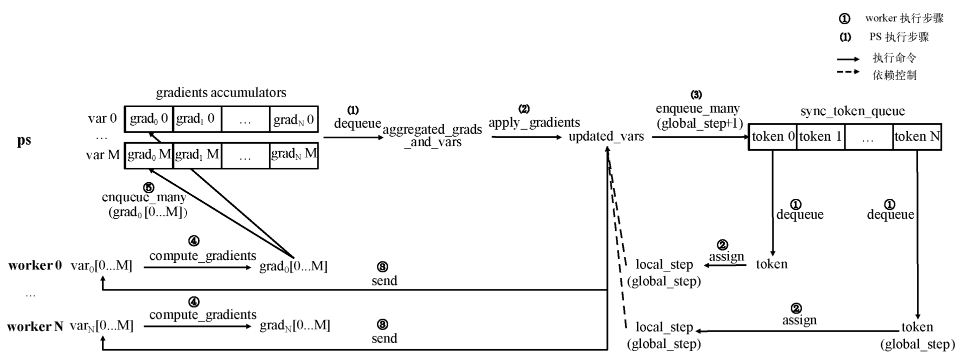


图5-5　同步优化器“全民参与”模式的工作流程

为了方便说明，我们假设模型参数的个数为 M，并行副本数和实际副本数均为 N，且每个worker都接收并使用全量共 M 个模型参数进行训练。在“全民参与”模式中，同步优化器涉及两个重要组件：梯度聚合器（gradients accumulator）和同步标记队列。

- **梯度聚合器**：存储梯度值的队列。以模型参数作为区分，每个模型参数拥有一个单独的队列。队列收集来自于不同 worker、根据不同批数据计算出的该模型参数对应的梯度值。如图5-5所示，梯度聚合器共包含 M 个队列，对应 M 个模型参数，每个队列收集来自于 N 个worker计算出的 N 个梯度值。
- **同步标记队列**：存储同步标记的队列。同步标记决定 worker 是否能够执行梯度计算任务。当队列中没有同步标记时，worker无法从PS获取更新后的模型参数，也就无法进行梯度计算。

图5-5展示的工作流程的执行实体是worker和PS，下面我们分别介绍它们在“全民参与”模式中的具体工作步骤。其中worker在“全民参与”模式中的工作步骤如下。

(1) 从同步标记队列中取出一个值为 `global_step`、表示全局训练步数的同步标记。

(2) 将同步标记的值赋予 worker 的本地训练步数（`local_step`）。

(3) 从 PS 获取最新的模型参数。

(4) 计算得到 M 个模型参数对应的 M 个梯度值。

(5) 将计算出的 M 个梯度值推送到 PS 上对应的 M 个模型参数的梯度队列中。

PS 在“全民参与”模式中的工作步骤如下。

(1) PS 上的梯度聚合器收集 worker 推送来的梯度值。当每个梯度队列收集到 N 份当前训练步（值为 `global_step`）的梯度值时，就求出该模型参数对应的梯度平均值。直到所有梯度队列都收集齐对应的梯度值后，最终出队得到包含 M 对 `{模型参数，对应梯度平均值}` 的聚合元组（`aggreagated_grads_and_vars`）。

(2) 将梯度平均值更新到对应的模型参数，得到更新后的模型参数。

(3) 向同步标记队列推送下一步训练需要的值为 `global_step+1` 的 N 个同步标记。

同步更新机制是一种基于全局同步时序更新模型参数的机制。在梯度聚合器和同步标记队列的配合下，同步优化器确保每一步训练时各个 worker 在不同批数据上计算出的梯度能够被统一聚合并做平均，然后一次性同步更新到 PS 的模型参数中。同步更新机制实现了真正的数据并行，它将单机无法处理的数据横向扩展到多个不同的 worker 上并行计算。所有参与计算的 worker 共同组成了一个拥有更强计算能力的虚拟计算设备，以此提升数据处理能力。

下面我们基于图 5-5 的工作流程，介绍同步更新机制的实现原理。梯度聚合器在收集梯度值时会校验当前梯度的版本号（应该被赋值为 `local_step`）。如果某个 worker 的 `local_step` 不等于 `global_step`（说明它们不属于同一个训练步，不符合同步更新的定义），那么梯度聚合器就会拒绝收集它推送的梯度值。直到收集够 N 份当前训练步的梯度后，PS 才更新对应的模型参数。计算能力强的 worker 完成梯度计算并将其推送到 PS 后，它可能阻塞在第(5)步。因为同步标记队列中的标记已被其他 worker 全部取出，所以计算能力强的 worker 只能等待其他 worker 完成当前步的执行步骤。直到当前训练步的模型参数更新后，PS 才会向同步标记队列入队新的同步标记。此时，所有 worker 再开始下一步训练。

同步更新机制依赖于同步标记队列的状态。每一步训练启动的前提是同步标记队列被填满了 N 个值为 `global_step` 的同步标记。在第一次训练开始前，因为 PS 没有更新模型参数，所以不会向同步标记队列入队标记。为此，我们需要执行一个初始化同步标记队列的操作——`sync_init_op`。它不依赖于模型参数的更新操作，而是直接向同步标记队列填充 N 个值为 0 的标记。于是，该初始化操作如同触发器一样开启了整个训练过程。

除了初始化同步标记队列外，表 5-4 列出了用户在训练分布式模型前需要执行的其他主要初始化操作。不过，如果用户使用 TensorFlow 的模型管理类——`Supervisor`，那么可以将这些初始化操作全部交给 `Supervisor` 并在其内部实现。

表 5-4 使用同步训练机制时的主要初始化操作

操作名称	常用变量名或简称	功能说明
opt.local_step_init_op	local_init_op	设置 local_step 的初始值
opt.chief_init_op	local_init_op	设置 global_step 的初始值
opt.ready_for_local_init_op	ready_for_local_init_op	为未初始化的 Variable 设置初始值
opt.get_init_tokens_op	sync_init_op	为同步标记队列入队初始值
tf.global_variables_initializer	init_op	为全局 Variable 设置初始值

注：opt 特指 tf.SyncReplicasOptimizer。

下面的代码给出了表 5-4 中初始化操作的典型使用方法：

```
# 使用同步训练机制
if FLAGS.sync_replicas:
  # 非 chief worker：为 local_step 设置初始值
  local_init_op = opt.local_step_init_op
  # chief worker：为 global_step 设置初始值
  if is_chief:
    local_init_op = opt.chief_init_op
  # 定义对未初始化变量设置初始值的操作
  ready_for_local_init_op = opt.ready_for_local_init_op

  # 定义启动同步标记队列的 QueueRunner 实例
  chief_queue_runner = opt.get_chief_queue_runner()
  # 定义对同步标记队列入队初始值的操作
  sync_init_op = opt.get_init_tokens_op()
# 定义对全局变量设置初始值的操作
init_op = tf.global_variables_initializer()
```

接下来，我们简要介绍“能者多劳”模式。对比“全民参与”和“能者多劳”两种模式的工作流程，有助于进一步理解数据并行的本质。数据并行解决的核心问题是增大单个训练步的批数据规模。当计算资源不足时，可以借助梯度聚合器和同步标记队列实现数据并行。图 5-6 展示了“能者多劳”模式的工作流程。

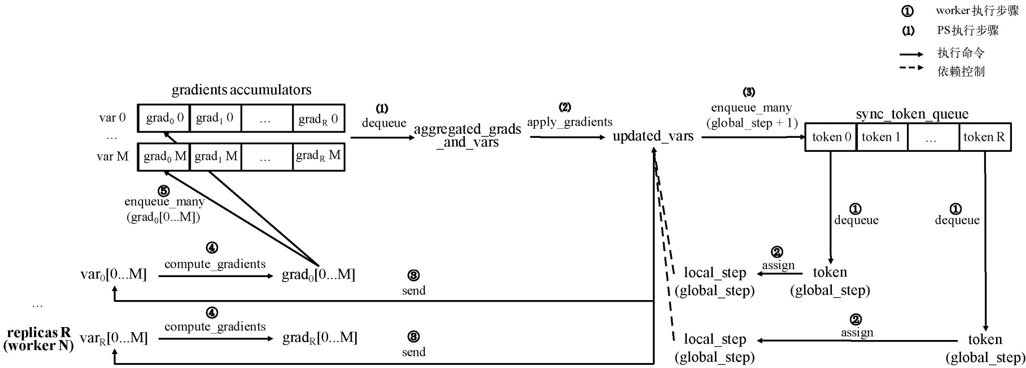


图 5-6 同步优化器“能者多劳”模式的工作流程

为了方便说明，我们仍然假设模型参数个数为 M，worker 任务数为 N。另设并行副本数为 R（$R > N$）。下面介绍“全民参与”模式下梯度聚合器和同步标记队列的功能变化。

- **梯度聚合器**：因为模型个数不变，所以我们仍然需要 M 个梯度队列来收集不同批数据上计算出的梯度值。但是，每个梯度队列需要收集的梯度份数为 R，大于 worker 任务数，这说明部分计算能力强的 worker 每一步都必须领取多批数据计算梯度。
- **同步标记队列**：存储 R 个同步标记，确保每一步训练时梯度聚合器都能够收集 R 份数据上并行计算出的梯度。计算能力强的worker在推送梯度到PS后，最多可以从同步标记队列中获取剩余的 R–N 个同步标记。

在学习了“全民参与”和“能者多劳”模式的工作流程后，读者可以自行思考“替补等位”模式的工作流程。

5.2.7　异步训练机制

使用 TensorFlow 异步训练分布式模型时，既不需要创建同步优化器实例，也不需要执行额外的初始化操作。异步训练机制的本质是将存储在 PS 上的模型参数共享给所有的 worker，每个 worker 计算出的梯度值都直接更新模型参数，不用等待其他 worker。程序不做梯度聚合器校验和梯度收集，也不需要从同步标记队列中获取同步标记。所有 worker 的训练行为与单机模型训练几乎一致，唯一不同在于模型参数需要从其他进程（PS）获取。当不同的 worker 同时进行参数更新和拉取操作时，TensorFlow 内部的锁机制保证模型参数的数据一致性。图 5-7 展示了异步训练的工作原理。

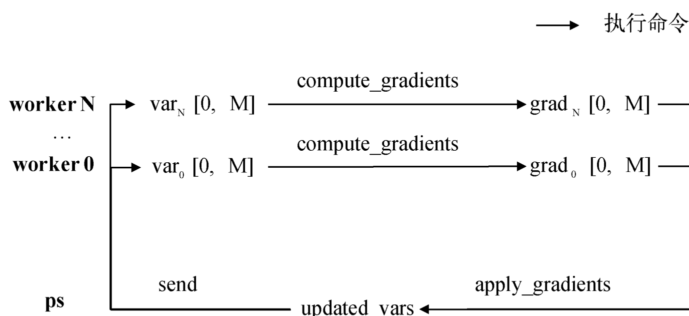


图 5-7　异步训练的工作原理

异步训练过程因为没有创建同步优化器实例，所以也没有并行副本数和实际副本数的概念。异步训练的好处是提高了资源利用率，它保证没有 worker 处于“替补”或空闲状态。所有的 worker 都可以持续不断地进行本地训练，不必担心任何的控制依赖问题。因此，异步训练机制下各 worker 的本地训练步数与全局训练步数也不相等，计算能力强的本地训练步数相对较大，计算能力弱的本地训练步数相对较小。

5.2.8 使用Supervisor管理模型训练

在使用 TensorFlow 训练模型的过程中，难免出现意外导致训练异常结束。为了提升模型训练过程的健壮性，TensorFlow 为用户提供了训练管理类——`Supervisor`。该类同时支持单机和分布式模型训练的管理，它以一种尽可能简单的使用方式为用户提供了以下 3 种功能。

- 在模型训练过程中，定期保存模型参数到 checkpoint 文件。
- 在重新启动训练程序时，从 checkpoint 文件中读取和恢复模型参数，并继续训练。
- 在异常发生时，处理程序关闭和异常退出，同时完成内存回收等清理工作。

`Supervisor` 将大部分定期操作和异常处理封装成接口。用户只需在初始化它时设置好相关参数，之后便不再需要做额外的逻辑控制和条件判断。`Supervisor` 本质上是对 3 个类——`Saver`、`SessionManager` 和 `Coordinator` 的封装，它们的功能如下。

- **`Saver`**：模型参数的存储和恢复。
- **`Coordinator`**：多线程服务的生命周期管理。
- **`SessionManager`**：单机和分布式会话的管理。

`Saver` 的功能与使用方法已在 4.2 节中介绍过。除了前述基本功能外，它还能够定期执行汇总操作，并将输出结果序列化保存到汇总事件文件中。这些文件可用于多种不同类型的可视化。我们将在下一章中详细介绍基于这一机制实现的 TensorBoard 可视化工具，现在先把目光聚焦在模型训练管理上。

`Coordinator` 主要负责监控训练程序中启动的多个线程是否运行正常。在模型训练过程中，任何服务抛出异常时都会向 `Supervisor` 报告。此时，`Coordinator` 会将程序的停止条件设置为 `True`，`Supervisor` 随即停止模型的训练，并清理工作现场（关闭会话、回收内存等）。其他服务检测到停止条件变为 `True` 后，便会停止运行，并关闭各自的线程。

如果说 `Saver` 和 `Coordinator` 的组合增强了模型训练的健壮性，那么 `SessionManager` 则为模型训练提供了实际的运行环境。`SessionManager` 帮助用户创建并管理单机或分布式会话，以便简化数据流图算法的生命周期维护逻辑。同时，它还负责将 checkpoint 文件中存储的模型参数恢复到会话加载的数据流图中。

使用 `Supervisor` 管理模型训练的典型流程如下。

(1) 创建一个 `Supervisor` 实例，向其构造方法传入 checkpoint 文件和汇总事件文件的存储目录（logdir）。

(2) 调用 `tf.train.Supervisor.managed_session` 方法，从 `Supervisor` 实例获取一个会话实例。

(3) 使用该会话执行训练操作，并在训练过程中检查停止条件，以确保模型训练的正确性。

用户在调用 `managed_session` 方法获取会话实例时，`Supervisor` 通过 `QueueRunner` 同时启

动了以下 3 个独立运行在各自线程中的标准服务。

- **检查点服务**：将数据流图中的模型参数定期保存到 logdir 下的 checkpoint 文件。如果用户在数据流图中正确添加了 `global_step`，那么 checkpoint 会根据 `global_step` 的值生成文件名。默认情况下，每 10 分钟执行一次。
- **汇总服务**：执行所有的汇总操作，并将输出结果追加到 logdir 下的汇总事件文件中。默认情况下，每 2 分钟执行一次。
- **步数计数器服务**：通过观察 `global_step` 的变化，记录当前执行的训练步数。同时，向汇总事件文件追加一条当前训练步执行时间的记录，汇总标记为 global_step/sec。默认情况下，每 2 分钟执行一次。

当训练正常结束或异常退出后，最新的 checkpoint 文件和汇总事件文件都会保存在 logdir 目录下。用户使用 `managed_session` 方法重新创建会话时，`Supervisor` 能够自动恢复 logdir 目录下最新的 checkpoint 文件，为数据流图中的模型参数赋予此前保存的值，然后继续训练。

下面首先给出使用 `Supervisor` 管理单机模型训练的简单代码实现：

```
# ……创建数据流图……
# 创建一个 Supervisor 实例
sv = tf.train.Supervisor(logdir="/my/training/directory")
# 获取一个进程内的会话实例，并完成模型参数的初始化
with sv.managed_session() as sess:
  # 初始化完成后，使用该会话训练模型
  while not sv.should_stop():
    sess.run(train_op)
```

接着给出使用 `Supervisor` 管理分布式模型训练的代码实现：

```
# ……创建数据流图……
server = tf.train.Server(cluster,
                         job_name=FLAGS.job_name,
                         task_index=FLAGS.task_index)
# 判断当前进程是否为 chief worker
is_chief = (FLAGS.task_index == 0)
# 创建一个 Supervisor 实例，将日志目录设置在共享文件系统上，
# 并指明当前进程是否为 chief worker
sv = tf.train.Supervisor(logdir="/shared_directory/...",
                         is_chief=is_chief)
# 在 TensorFlow 集群的指定 server 上获取一个会话实例
# 如果当前进程是 chief worker，那么初始化全局的模型参数
# 如果当前进程不是 chief worker，那么等待 chief worker 完成初始化
with sv.managed_session(server.target) as sess:
  # 初始化完成后，使用该会话训练模型
  while not sv.should_stop():
    sess.run(train_op)
```

如果当前进程是 chief worker，那么 `Supervisor` 启动的服务与单机模型的情况一致。如果当前进程不是 chief worker，那么 `Supervisor` 在实例化会话后，将等待 chief worker 完成全局模型参数的初始化。直到 chief worker 初始化完成后，非 chief worker 才会得到一个可用的分布式会话

对象，进而开始训练模型。

当 TensorFlow 集群中某个 worker 异常退出并重启程序时，managed_session 方法会检查全局的模型参数是否已完成初始化。如果是，那么该方法将直接返回一个可用的分布式会话，并继续训练；如果否，那么 chief worker 将重新初始化全局模型参数，非 chief worker 则会停止训练并等待模型参数初始化完成。

5.2.9　分布式同步训练的最佳实践

本节结合前几节学习的内容，通过丰富的注释，从代码实现层面解析 TensorFlow 分布式程序编程框架的工作流程和使用方法：

```
"""5.2_best_practice.py"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import math
import sys
import tempfile
import time

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data


flags = tf.app.flags
flags.DEFINE_string("data_dir", "/tmp/mnist-data",
                    "Directory for storing mnist data")
flags.DEFINE_string("train_dir", "/tmp/mnist-log",
                    "Directory for storing checkpoint and summary files")
flags.DEFINE_integer("task_index", None,
                     "Worker task index, should be >= 0. task_index=0 is "
                     "the master worker task the performs the variable "
                     "initialization ")
flags.DEFINE_integer("replicas_to_aggregate", None,
                     "Number of replicas to aggregate before parameter update"
                     "is applied (For sync_replicas mode only; default: "
                     "num_workers)")
flags.DEFINE_integer("hidden_units", 100,
                     "Number of units in the hidden layer of the NN")
flags.DEFINE_integer("train_steps", 200,
                     "Number of (global) training stePS to perform")
flags.DEFINE_integer("batch_size", 100, "Training batch size")
flags.DEFINE_float("learning_rate", 0.01, "Learning rate")
flags.DEFINE_boolean("sync_replicas", False,
                     "Use the sync_replicas (synchronized replicas) mode, "
                     "wherein the parameter updates from workers are aggregated "
                     "before applied to avoid stale gradients")
flags.DEFINE_string("ps_hosts","localhost:2222",
                    "Comma-separated list of hostname:port pairs")
```

```
flags.DEFINE_string("worker_hosts", "localhost:2223,localhost:2224",
                    "Comma-separated list of hostname:port pairs")
flags.DEFINE_string("job_name", None,"job name: worker or PS")

FLAGS = flags.FLAGS


IMAGE_PIXELS = 28


def main(unused_argv):
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

  if FLAGS.job_name is None or FLAGS.job_name == "":
    raise ValueError("Must specify an explicit `job_name`")
  if FLAGS.task_index is None or FLAGS.task_index =="":
    raise ValueError("Must specify an explicit `task_index`")

  # 解析 PS 和 worker 的主机名、端口列表
  PS_spec = FLAGS.ps_hosts.split(",")
  worker_spec = FLAGS.worker_hosts.split(",")

  # 计算 worker 的数量
  num_workers = len(worker_spec)

  cluster = tf.train.ClusterSpec({
      "PS": PS_spec,
      "worker": worker_spec})

  # 如果是 PS，直接启动服务，并开始监听 worker 发起的请求
  if FLAGS.job_name == "PS":
    server.join()

  # 根据 TensorFlow 集群的定义和当前设备的信息，放置对应的模型参数和计算操作
  with tf.device(
      tf.train.replica_device_setter(
          worker_device=worker_device,
          PS_device="/job:PS/cpu:0",
          cluster=cluster)):
      global_step = tf.Variable(0, name="global_step", trainable=False)

      # 隐层模型参数
      hid_w = tf.Variable(
          tf.truncated_normal(
              [IMAGE_PIXELS * IMAGE_PIXELS, FLAGS.hidden_units],
              stddev=1.0 / IMAGE_PIXELS),
          name="hid_w")
      hid_b = tf.Variable(tf.zeros([FLAGS.hidden_units]), name="hid_b")

      # softmax 层模型参数
      sm_w = tf.Variable(
          tf.truncated_normal(
              [FLAGS.hidden_units, 10],
              stddev=1.0 / math.sqrt(FLAGS.hidden_units)),
```

```
        name="sm_w")
    sm_b = tf.Variable(tf.zeros([10]), name="sm_b")

    # 根据任务编号放置对应的placeholder
    x = tf.placeholder(tf.float32, [None, IMAGE_PIXELS * IMAGE_PIXELS])
    y_ = tf.placeholder(tf.float32, [None, 10])
    # tf.nn.xw_plus_b即为matmul(x, w) + b
    hid_lin = tf.nn.xw_plus_b(x, hid_w, hid_b)
    # 使用relu作为激活函数，hid作为隐层输出
    hid = tf.nn.relu(hid_lin)
    # 定义softmax层的输出y，即推理计算出的标签值
    y = tf.nn.softmax(tf.nn.xw_plus_b(hid, sm_w, sm_b))
    # 使用交叉熵评估两个概率分布间的相似性
    # 因为概率取值范围为[0, 1]，为避免出现无意义的log(0)，故将y值裁剪到[1e-10, 1.0]
    cross_entropy = -tf.reduce_sum(y_ * tf.log(tf.clip_by_value(y, 1e-10, 1.0)))
    # 使用Adam做最优化求解
    opt = tf.train.AdamOptimizer(FLAGS.learning_rate)

  # 如果使用同步训练机制
  if FLAGS.sync_replicas:
    # 如果用户没有输入并行副本数，则令其等于worker任务数
    if FLAGS.replicas_to_aggregate is None:
      replicas_to_aggregate = num_workers
    # 如果用户输入了并行副本数，则赋值为命令行解析的并行副本数
    else:
      replicas_to_aggregate = FLAGS.replicas_to_aggregate
    # 创建同步优化器实例，负责计算梯度和更新模型参数
    opt = tf.train.SyncReplicasOptimizer(
        opt,
        replicas_to_aggregate=replicas_to_aggregate,
        total_num_replicas=num_workers,
        name="mnist_sync_replicas")
  # 单步训练操作，即利用同步优化器最优化交叉熵
  train_op = opt.minimize(cross_entropy, global_step=global_step)

  # 使用同步训练机制
  if FLAGS.sync_replicas:
    # 其他worker：为local_step设置初始值
    local_init_op = opt.local_step_init_op
    # chief worker：为global_step设置初始值
    if is_chief:
      local_init_op = opt.chief_init_op
    # 定义为未初始化的Variable设置初始值的操作
    ready_for_local_init_op = opt.ready_for_local_init_op

    # 定义启动同步标记队列的QueueRunner实例
    chief_queue_runner = opt.get_chief_queue_runner()
    # 定义同步标记队列入队初始值的操作
    sync_init_op = opt.get_init_tokens_op()
  # 定义全局Variable设置初始值的操作
  init_op = tf.global_variables_initializer()
  # 判断当前是否为chief worker的任务进程
  is_chief = (FLAGS.task_index == 0)
```

```
# 使用同步训练机制，传入本地初始化相关操作
if FLAGS.sync_replicas:
  sv = tf.train.Supervisor(
      is_chief=is_chief,
      logdir=FLAGS.train_dir,
      init_op=init_op,
      local_init_op=local_init_op,
      ready_for_local_init_op=ready_for_local_init_op,
      recovery_wait_secs=1,
      global_step=global_step)
# 使用异步更新机制，各 worker 独自训练，与单机模型一致
else:
  sv = tf.train.Supervisor(
      is_chief=is_chief,
      logdir=FLAGS.train_dir,
      init_op=init_op,
      recovery_wait_secs=1,
      global_step=global_step)
# 配置分布式会话：
#     在没有可用的 GPU 时，将操作放置到 CPU
#     不打印设备放置信息
#     过滤未绑定在 PS 和 worker 上的操作
sess_config = tf.ConfigProto(
    allow_soft_placement=True,
    log_device_placement=False,
    device_filters=["/job:PS", "/job:worker/task:%d" % FLAGS.task_index])

# 如果是 chief worker，则初始化所有 worker 的分布式会话
if is_chief:
  print("Worker %d: Initializing session..." % FLAGS.task_index)
# 如果是其他 worker，则等待 chief worker 返回的会话
else:
  print("Worker %d: Waiting for session to be initialized..." %
        FLAGS.task_index)

sess = sv.prepare_or_wait_for_session(server.target, config=sess_config)

print("Worker %d: Session initialization complete." % FLAGS.task_index)
# 如果是同步更新模式，并且当前进程为 chief worker
if FLAGS.sync_replicas and is_chief:
  # 初始化同步标记队列
  sess.run(sync_init_op)
  # 通过 QueueRunner 启动 3 个线程，并运行各自的标准服务
  sv.start_queue_runners(sess, [chief_queue_runner])

# 记录并打印训练开始前的时间
time_begin = time.time()
print("Training begins @ %f" % time_begin)
# 将 local_step 赋值为 0
local_step = 0
while True:
  # 填充训练数据
  batch_xs, batch_ys = mnist.train.next_batch(FLAGS.batch_size)
  train_feed = {x: batch_xs, y_: batch_ys}
```

```
            # 执行单步训练操作
            _, step = sess.run([train_op, global_step], feed_dict=train_feed)
            local_step += 1
            # 记录并打印完成当前单步训练所需的时间
            now = time.time()
            print("%f: Worker %d: training step %d done (global step: %d)" %
                  (now, FLAGS.task_index, local_step, step))
            # 如果当前超过最大训练步数，退出训练循环
            if step >= FLAGS.train_steps:
              break
          # 记录并打印训练结束的时间
          time_end = time.time()
          print("Training ends @ %f" % time_end)
          # 总训练时间为两者的时间差
          training_time = time_end - time_begin
          print("Training elapsed time: %f s" % training_time)

          # 填充验证数据
          val_feed = {x: mnist.validation.images, y_: mnist.validation.labels}
          # 在验证数据集上计算模型的交叉熵
          val_xent = sess.run(cross_entropy, feed_dict=val_feed)
          print("After %d training step(s), validation cross entropy = %g" %
                (FLAGS.train_steps, val_xent))


if __name__ == "__main__":
  tf.app.run()
```

5.3　小结

本章以用户编程视角切入，结合宏观理论与最佳实践，全方位剖析了单机程序与分布式程序的典型编程框架。TensorFlow 的编程框架兼具灵活性和包容性，通过组合大量细粒度的 API，允许用户自由地设计多样化的深度学习应用程序。单机编程框架提供对“单进程、多设备”计算模式的支持，以数据流图和会话抽象为用户提供直观易用的深度学习开发环境。分布式编程框架提供同步优化器和 Supervisor 机制，保证模型训练过程的可用性和健壮性。这有助于用户更加专注算法模型的设计与实现，从而不需要在系统层面的问题上消耗太多精力。TensorFlow 根据数据流图的构建模式和模型参数的更新机制，将常见的数据并行模式划分为 4 种不同的类型。读者应该熟练掌握这 4 种数据并行模式，这有助于理解通用的机器学习程序设计思想。

第 6 章 TensorBoard 可视化工具

在算法设计过程中，开发者往往需要剖析神经网络的结构和数据流图的流程。在模型训练过程中，用户也常常需要关注参数的变化趋势和模型的实时效果。晦涩的代码和单调的日志难以满足上述需求，图形界面工具的出现成为必然。TensorBoard 是 TensorFlow 项目组开发的深度学习可视化工具，它通过展示直观的图形，能够有效地辅助深度学习程序的开发者和使用者理解算法模型及其工作流程。本章首先介绍 TensorBoard 的可视化效果和典型使用方法，然后依次讲解使用 TensorBoard 可视化数据流图、可视化学习过程和可视化高维数据的最佳实践。同时，我们会在各节穿插介绍与 TensorBoard 密切相关的编程接口——`tf.summary` 模块的组成结构和工作原理。

6.1 概述

TensorBoard 的实现形态为 Web 应用程序，这为提供分布式、跨系统的图形界面服务带来了便利。本节通过直观的插图给出典型的用例，向读者展示 TensorBoard 的基本功能。

TensorBoard Web 界面的顶部菜单栏列出了 7 个功能面板的链接，这些面板分别用于展示不同的可视化对象。其中，SCALARS 面板展示标量值随时间变化的关系图，IMAGES 和 AUDIO 面板分别展示图像和音频数据，GRAPHS 面板展示数据流图，DISTRIBUTIONS 和 HISTOGRAMS 面板分别展示向量值的数据分布和统计信息，EMBEDDINGS 面板则用于展示降维后的高维数据。

图 6-1 展示了使用 TensorBoard 可视化 MNIST softmax 模型对应的数据流图。在使用 TensorFlow 开发模型的过程中，用户设计的数据流图往往十分复杂，其中可能包含少则成几百条、多则上万条数据流或控制流。如果没有一个合适的工具展示模型对应的数据流图，那么调试的效率将会非常低下。为此，TensorBoard 的 GRAPHS 面板实现了数据流图的可视化功能，能够分级显示子图和子节点，这可以大大提升静态调试模型结构的效率。

图 6-2 展示了使用 SCALARS 面板可视化模型训练过程中准确率和交叉熵随时间的变化。随着训练步数的增加，准确率不断上升，交叉熵不断减小，表示模型在训练集上的预测结果越来越准确。为了更好地理解模型的训练过程，我们除了可以可视化准确率和交叉熵这类度量指标外，

也可以将模型参数张量中所有元素的数据分布和统计信息可视化。事实上，除了模型的训练过程，还可以将模型的验证过程和测试过程中的度量指标和模型参数可视化。我们将其统一称为可视化学习过程。

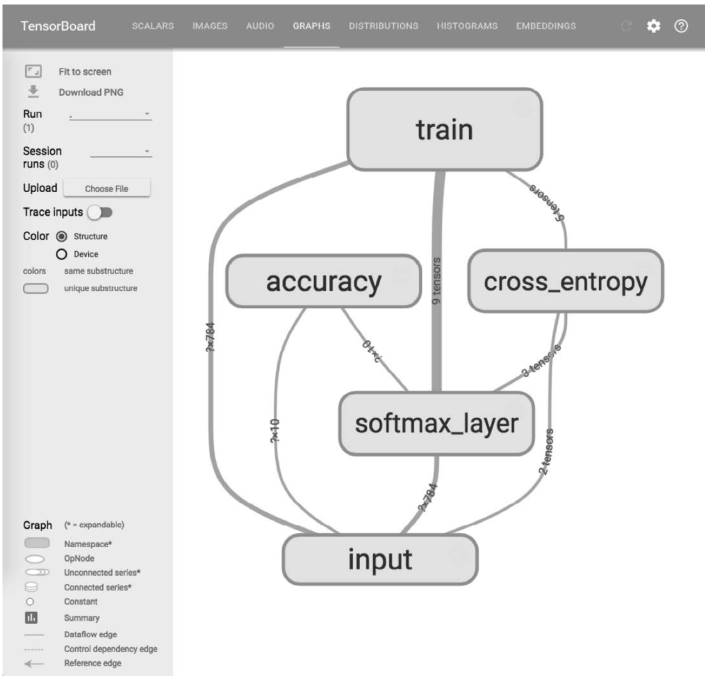


图 6-1　使用 TensorBoard 可视化数据流图

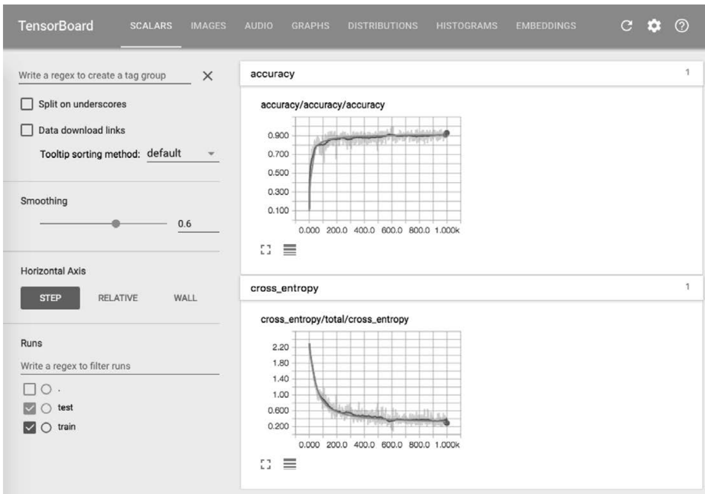


图 6-2　使用 TensorBoard 可视化学习过程

在深度学习和机器学习任务中，可视化高维数据（如模型参数）常常是个令人头疼的问题，好在 TensorBoard 能够提供这项能力。图 6-3 给出了一个示例。在本例中，我们首先使用 T-SNE 方法将 MNIST 测试集中的手写体图像数据转换成的高维数据降到三维，然后将每张图像的三维数据对应生成三维直角坐标系中的一个点，最终展示出总共 10000 个图像数据在三维直角坐标系中的分布情况。不难发现，数字相同的手写体图像大多分布在一块空间。随着训练步数的增加，模型分类的效果也会越来越好。通过可视化模型输出的高维数据，我们可以直观地感受模型的分类效果。

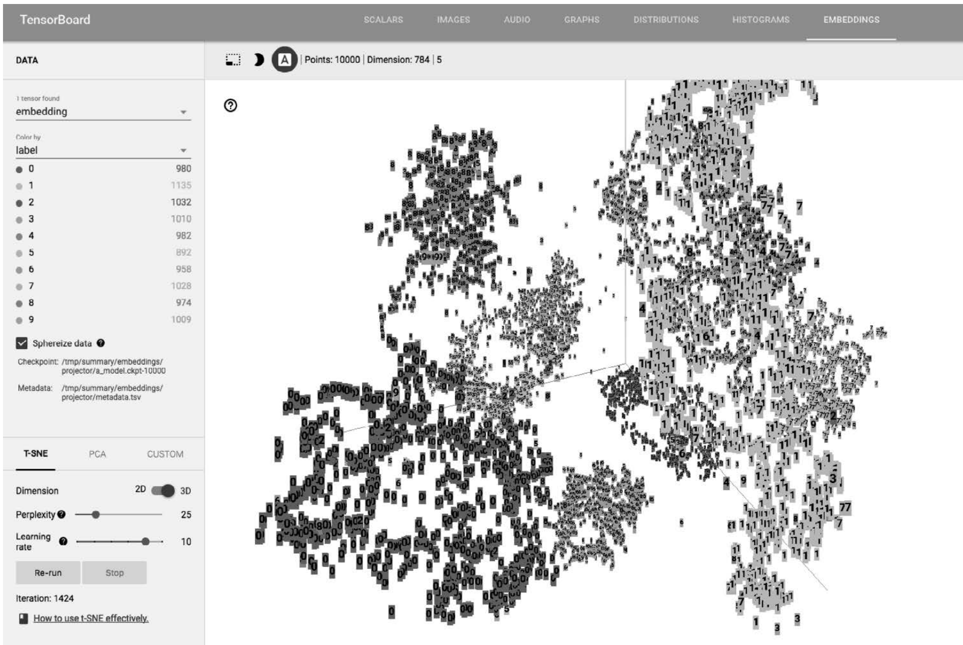


图 6-3　使用 TensorBoard 可视化高维数据

TensorBoard使用流程

从上面展示的 3 类对象来看，TensorBoard 的可视化效果的确不错。事实上，TensorBoard 在不同面板下展示的对象数据均来自于 TensorFlow 程序生成的序列化数据，有效地输出这些数据是使用 TensorBoard 的关键所在。下面我们简要介绍 TensorBoard 的使用流程。

图 6-4 给出了 TensorBoard 的典型使用流程。我们想要可视化的数据是数据流图和操作输出的张量，它们只有在会话中加载或执行后才能获取。这里先不细究如何获取到序列化的数据流图和张量数据，假设它们已经就绪。接下来，用户需要使用 `FileWriter` 实例将这些数据写入事件文件。最后，启动 TensorBoard 程序，加载事件文件中的序列化数据，从而可以在各个面板中展示对应的可视化对象。

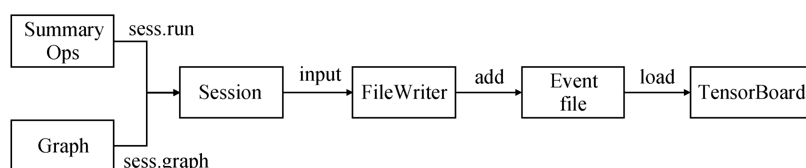


图 6-4 TensorBoard 的使用流程

这里使用的 `FileWriter` 实例和汇总操作（summary ops）均属于 `tf.summary` 模块，而这个模块贯穿 TensorBoard 的整个使用流程。因此，这里简单介绍一下这个模块，它的主要功能是获取和输出模型相关的序列化数据。如图 6-5 所示，它的核心部分由一组汇总操作以及 `FileWriter`、`Summary` 和 `Event` 这 3 个类组成。

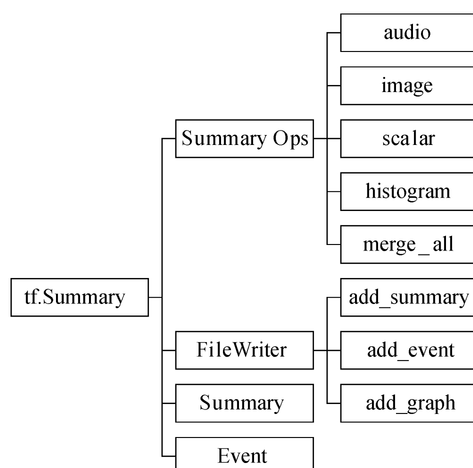


图 6-5 `tf.summary` 模块的组成结构

汇总操作主要负责获取数据流图上的张量，包括用户定义的度量指标、模型参数、音频和图像数据等。汇总操作与计算操作输出的张量不同，前者存储的是符合 TensorFlow 项目组预定义数据结构的序列化数据（Protocol Buffers），后者包含的是具体的张量值。`merge_all` 方法将所有的汇总操作汇聚在一起，生成了一个聚集操作 `merged`。它是所有汇总操作的后置操作，只要用户执行了 `merged` 操作，就等同于执行了所有的汇总操作。

`FileWriter` 类负责向事件文件写入序列化数据，包括汇总数据、事件数据和数据流图等。其中，汇总数据符合 tensorflow/core/framework/summary.proto 文件定义的数据结构，表示一组将被可视化的具名数值，包括图像数据、音频数据等；事件数据符合 tensorflow/core/util/event.proto 文件定义的数据结构，表示在会话中执行操作时产生的事件信息，包括时间戳、全局步数等；数据流图符合 tensorflow/core/framework/graph.proto 文件定义的数据结构，表示当前会话加载的默认数据流图，包括图上所有的节点和有向边，以及图的版本号等。图 6-5 中的 `Summary` 类便是 summary.proto 文件定义的汇总数据，而 `Event` 类便是 event.proto 文件定义的事件数据。

6.2　可视化数据流图

在使用 TensorFlow 创建模型的过程中，通常难以确保代码生成的数据流图与自己设想的拓扑结构完全一致。当模型输出无效结果时，往往需要调试程序。对于网络结构简单的小模型来说，或许可以通过阅读代码和手绘数据流图查错。但是对于深度神经网络这类复杂模型来说，手绘数据流图不仅效率低下，而且容易产生错误。因此，我们需要学会使用 TensorBoard 可视化数据流图。本节首先介绍使用名字作用域定义抽象节点的方法，接着介绍可视化 MNIST softmax 模型的数据流图的最佳实践。最后，详细介绍 summary.proto 和 event.proto 中定义的汇总数据和事件数据的数据结构，以及 `FileWriter` 类的工作原理。

6.2.1　名字作用域与抽象节点

经过前面几章的学习，我们知道根据功能不同，数据流图节点可以分为计算节点、存储节点、数据节点和汇总节点。本节引入一个新概念——抽象节点。与之前介绍的节点相比，它并不代表一个具体的操作，而是代表一组特定操作的集合。将同一层网络或具有相同功能的操作整合为一个抽象节点，可以简化数据流图的网络结构。

下面我们以 MNIST softmax 模型的数据流图为例，对比使用抽象节点前后网络结构的复杂度。如图 6-6 所示，左图是直接使用 TensorBoard 可视化的 MNIST softmax 模型的数据流图，整幅图看起来比较凌乱，并且缺少后向图的部分。右图是使用抽象节点分别整合了模型输入、softmax 层、准确率、交叉熵和训练参数的后向图的数据流图。右图整体看起来非常简洁，而且各个模块的功能和前后依赖也很清晰。左右对比，不难发现使用抽象节点的好处。

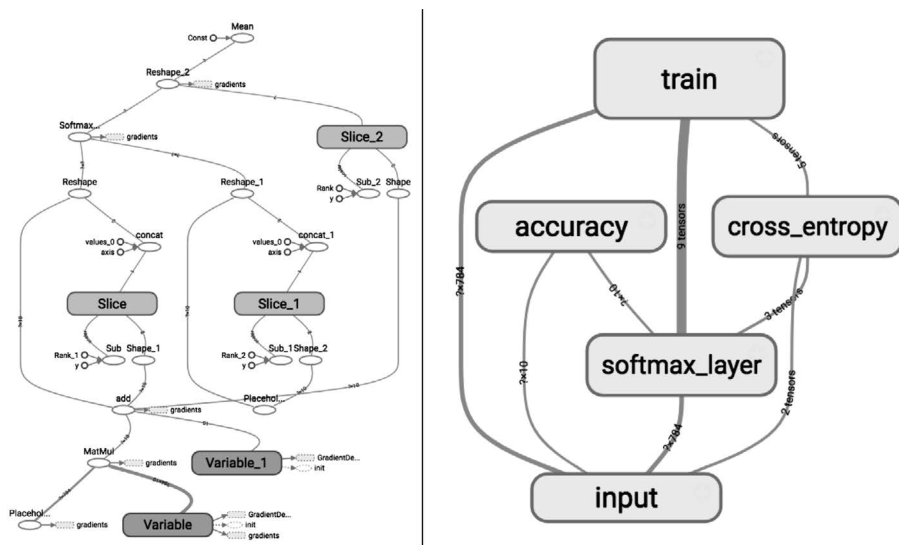


图 6-6　MNIST softmax 模型的数据流图（左）和使用抽象节点后的数据流图（右）

虽然 MNIST softmax 模型仅包含一层 softmax 网络层，但是如果不做任何处理直接可视化，它的数据流图网络结构仍稍显复杂。试想如果直接可视化深度神经网络模型，如 VGG-NET、ResNet 等包含成千上万个节点的复杂模型，可能就会陷入节点间的复杂网络关系，难以迅速从高层的视角定位数据流图网络结构上的问题。因此，针对可视化深度神经网络模型的场景，我们应该尽可能使用抽象节点整合相同的网络层或功能模块。抽象节点是数据流图上的子图，其内部展开后并没有丢失节点间的拓扑关系。相比直接可视化数据流图，它的层次化结构更清晰明了。

下面介绍使用名字作用域定义数据流图上抽象节点的方法。相比只作用于存储节点的变量作用域，名字作用域更通用，因为它对计算、存储和数据节点都有效。同一名字作用域下的所有节点继承相同的名字前缀，在数据流图上体现为该作用域下的所有节点汇集成了同一个节点，如图 6-6 中的右图所示。

我们使用 `tf.name_scope` 方法创建名字作用域。下面的代码定义了图 6-6 右图中的 softmax_layer 抽象节点，它的内部包括图 6-6 左图中的 add、MatMul、Variable 和 Variable_1 节点：

```
with tf.name_scope('softmax_layer'):
  weights = tf.Variable(tf.zeros([784, 10]))
  biases = tf.Variable(tf.zeros([10]))
  y = tf.matmul(x, weights) + biases
```

虽然图 6-6 右图的整体结构简洁，但是一旦展开抽象节点，就会发现其内部仍然比较凌乱，如图 6-7 左图所示。

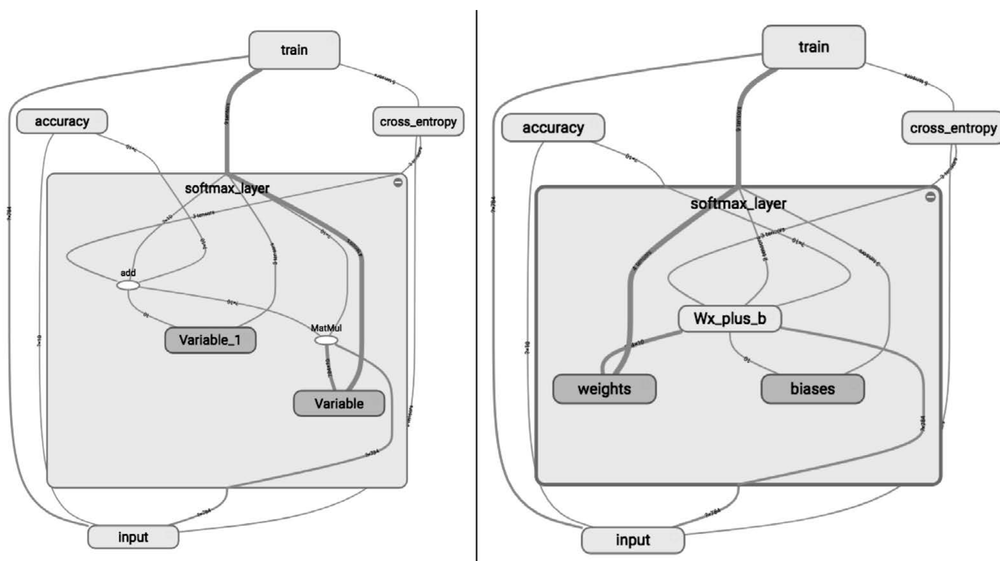


图 6-7 一层结构的 softmax_layer 抽象节点（左）和两层结构的 softmax_layer 抽象节点（右）

既然如此，我们需要进一步学习如何更优雅地构造数据流图的层次化结构。利用 `tf.name_scope` 方法支持嵌套定义名字作用域的特点，下面的代码定义了图 6-7 右图的 softmax_layer 抽象

节点，其内部 3 个节点的完整名称分别是 softmax_layer/weights、softmax_layer/biases 和 softmax_layer/Wx_plus_b：

```
with tf.name_scope('softmax_layer'):
  with tf.name_scope('weights'):
    weights = tf.Variable(tf.zeros([784, 10]))
  with tf.name_scope('biases'):
    biases = tf.Variable(tf.zeros([10]))
  with tf.name_scope('Wx_plus_b'):
    y = tf.matmul(x, weights) + biases
```

在 TensorBoard 的 GRAPHS 功能面板下，我们可以点击数据流图中的+按钮展开任意的抽象节点。相反，我们可以点击数据流图中的–按钮折叠任意的抽象节点。通过定义恰当的名字作用域，数据流图的网络结构能够变得更加清晰。当创建复杂的神经网络模型时，读者有必要为各个节点添加合适的名字作用域。

6.2.2　可视化数据流图的最佳实践

MNIST softmax 模型作为经典的手写体数字识别模型，不仅网络结构简单，而且识别正确率也不错，故常作为神经网络模型入门的最佳代表。本节以 MNIST softmax 模型为例，介绍使用 TensorBoard 可视化其数据流图的步骤和方法。

根据图 6-4 展示的 TensorBoard 典型使用流程，我们将可视化数据流图简化为以下 3 个步骤：

(1) 创建数据流图；

(2) 创建 `FileWriter` 实例；

(3) 启动 TensorBoard 程序。

下面详细介绍各个步骤的实现方法。

(1) 创建数据流图，并使用名字作用域将同一网络层或相同功能模块的节点定义为同一个抽象节点，如输入模块、softmax 网络层、交叉熵、优化器和准确率。同时，对于同一个抽象节点内部包含多个节点的情况，可以嵌套使用名字作用域，使得数据流图的网络结构更加清晰：

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input_data
# 获取 MNIST 数据集
mnist = input_data.read_data_sets('/tmp/data/mnist', one_hot=True)
# 输入模块
with tf.name_scope('input'):
  x = tf.placeholder(tf.float32, [None, 784], name='x-input')
  y_ = tf.placeholder(tf.float32, [None, 10], name='y-input')
# softmax 网络层
with tf.name_scope('softmax_layer'):
```

```
    with tf.name_scope('weights'):
      weights = tf.Variable(tf.zeros([784, 10]))
    with tf.name_scope('biases'):
      biases = tf.Variable(tf.zeros([10]))
    with tf.name_scope('Wx_plus_b'):
      y = tf.matmul(x, weights) + biases
  # 交叉熵
  with tf.name_scope('cross_entropy'):
    diff = tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y)
    with tf.name_scope('total'):
      cross_entropy = tf.reduce_mean(diff)
      tf.summary.scalar('cross_entropy', cross_entropy)
  # 优化器
  with tf.name_scope('train'):
    train_step = tf.train.AdamOptimizer(0.001).minimize(
        cross_entropy)
  # 准确率
  with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
      correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
    with tf.name_scope('accuracy'):
      accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

(2) 创建 `FileWriter` 实例，并将第 (1) 步中创建的数据流图写入事件文件。`FileWriter` 构造方法的第一个参数 `logdir` 是必须设置的，它表示 `FileWriter` 实例创建的事件文件的目录。由于同一个模型可能训练多次，`FileWriter` 会自动为每次训练创建各自的事件文件。当我们向 `FileWriter` 实例的构造方法传入数据流图时，它也会自动将该数据流图写入事件文件中：

```
  # 创建交互式会话
  sess = tf.InteractiveSession()
  # 创建 FileWriter 实例，并传入当前会话加载的数据流图
  writer = tf.summary.FileWriter('/tmp/summary/mnist', sess.graph)
  # 初始化全局变量
  tf.global_variables_initializer().run()
  # 关闭 FileWriter 的输出流
  writer.close()
```

(3) 启动 TensorBoard 程序，并设置 `logdir` 参数为第 (2) 步中定义的事件文件目录。当同一个模型训练多次后，TensorBoard 可以将该目录下的所有事件文件的数据组合在一起。在 TensorBoard 的 Web 界面上呈现时，用户只会看到数据的连续变化，而不会感觉到可视化的结果是由多个事件文件组成的。因为数据流图是相对静态的，所以不会受到多个事件文件的影响。相关代码如下：

```
  $ tensorboard --logdir=/tmp/summary/mnist
  Starting TensorBoard 41 on port 6006
```

TensorBoard 程序默认使用 6006 端口。启动程序后，在浏览器地址栏中输入 `<http://localhost:6006>`，然后在菜单栏中选择 GRAPHS 功能面板，就可以看到如图 6-7 右图所示的 MNIST softmax 模型的数据流图。

6.2.3 扩展阅读：汇总数据和事件数据

前面已经多次提到汇总数据、事件数据及其数据结构所在的文件 summary.proto 和 event.proto，

本节旨在说明它们各自的定义以及两者之间的关系。

1. 汇总数据与 summary.proto

汇总数据是 `tf.summary.Summary` 类或其内嵌类的实例。`tf.summary.Summary` 包含 3 个内嵌类——`Image`、`Audio` 和 `Value`，它们分别表示图像、音频和具名数值这 3 种典型数据。这里摘取 summary.proto 文件的代码，展示 `tf.summary.Summary` 类及其 3 个内嵌类的定义：

```
message Summary {
  // 图像数据
  message Image {
    // 图像大小
    int32 height = 1;
    int32 width = 2;
    // 颜色空间的有效值：
    //   1 - grayscale 灰度图
    //   2 - grayscale + alpha 带透明度的灰度图
    //   3 - RGB 彩色图
    //   4 - RGBA 带透明度的彩色图
    //   5 - DIGITAL_YUV 彩色图
    //   6 - BGRA 32bit 位图
    int32 colorspace = 3;
    // 编码后的图像数据
    bytes encoded_image_string = 4;
  }
  // 音频数据
  message Audio {
    // 音频采样率，单位为 Hz
    float sample_rate = 1;
    // 音频通道数
    int64 num_channels = 2;
    // 音频的总帧数（每通道中的采样数）
    int64 length_frames = 3;
    // 编码后的音频数据
    bytes encoded_audio_string = 4;
    // RFC 2045 格式的数据类型描述，如 "audio/wav"
    string content_type = 5;
  }
  // 具名数值
  message Value {
    // 输出该汇总数据的节点名称，如 "some_op"
    string node_name = 7;
    // 数据标签，如 "op_name:value_name"。
    // 其中，"op_name"通过层次结构能够设置操作所属的集合
    string tag = 1;

    // 数据标签对应的值
    oneof value {
      float simple_value = 2;
      bytes obsolete_old_style_histogram = 3;
      Image image = 4;
      HistogramProto histo = 5;
      Audio audio = 6;
```

```
      TensorProto tensor = 8;
    }
  }
  // 该汇总数据包含的具名数值数组
  repeated Value value = 1;
}
```

2. 事件数据与 event.proto

事件数据是 `tf.summary.Event` 类的实例。事件数据表示在会话中执行操作时产生的事件信息，包括时间戳、全局步数，以及以 oneof（多选一）方式定义的具体事件信息。对于一个特定的事件实例，事件信息只能从以下字段中选择一种：文件版本号、编码后的数据流图、汇总数据、TensorBoard 日志、会话状态、元数据和编码后的元数据流图——它们分别代表相应类型对象的创建事件。下面是我们从 event.proto 文件中摘取出 `tf.summary.Event` 类的定义：

```
message Event {
  // 时间戳
  double wall_time = 1;
  // 全局步数
  int64 step = 2;
  oneof what {
    // 文件版本号
    string file_version = 3;
    // 编码后的数据流图
    bytes graph_def = 4;
    // 汇总数据
    Summary summary = 5;
    // 由 Python 的 tensorboard_logging 模块生成的日志
    LogMessage log_message = 6;
    // 会话日志，以便在程序崩溃后重启会话
    SessionLog session_log = 7;
    // 调用 session.run()后返回的元数据
    TaggedRunMetadata tagged_run_metadata = 8;
    // 编码后的元数据流图
    bytes meta_graph_def = 9;
  }
}
```

可以看出，只要补充了时间戳和全局步数，我们就可以将汇总数据转换为事件数据。同理，数据流图、会话日志等也可以转换成事件数据。`FileWriter` 实例写入的序列化数据其实就是事件数据，只不过具体事件可能略有不同。下面我们介绍 `FileWriter` 的工作原理。

6.2.4　扩展阅读：揭秘tf.summary.FileWriter工作原理

根据前面的推论，我们认为 `FileWriter` 向事件文件中写入的是事件数据，那么汇总数据和其他的序列化数据一定得转换为事件数据。此外，因为训练过程会产生大量数据，所以 `FileWriter` 还得支持并行地更新事件文件。事实上，`FileWriter` 的确做到了这两点。我们从定义 `FileWriter` 类的 tensorflow/python/summary/writer/writer.py 文件中摘取出它的构造方法，如下所示：

```
class FileWriter(SummaryToEventTransformer):
  def __init__(self,
               logdir,
               graph=None,
               max_queue=10,
               flush_secs=120,
               graph_def=None,
               filename_suffix=None):
    # event_writer 是 EventFileWriter 的实例对象
    event_writer = EventFileWriter(logdir, max_queue, flush_secs)
    super(FileWriter, self).__init__(event_writer, graph, graph_def)
```

FileWriter 类继承自 SummaryToEventTransformer 类。FileWriter 实例包含 event_writer 成员对象，后者是 EventFileWriter 的实例，提供向事件文件中写入事件数据的能力。因此，FileWriter 内部实际调用的是 event_writer 成员的方法来完成事件文件的创建和更新。下面我们介绍构造方法中的输入参数。logdir 和 graph 分别指存储事件文件的目录和会话中加载的数据流图。max_queue 是缓存事件数据的队列大小。flush_secs 是定期刷新事件数据缓存队列的时间，单位为秒。因为 FileWriter 支持多线程并行更新事件文件，所以它和输入流水线一样需要多个缓存队列来提升效率。graph_def 是一个被遗弃的参数，现在直接使用 graph 参数替代它。FileWriter 构造方法仍然保留该参数，是为了兼容基于低版本 TensorFlow 开发的程序。filename_suffix 参数设置事件文件的名称后缀。

当我们创建 FileWriter 实例时，它的构造方法会在 logdir 目录下创建一个事件文件。对于用户显式传入 graph 参数的情况，因为 FileWriter 的构造函数内部调用了基类 SummaryToEventTransformer 的构造方法，所以后者将会调用 add_graph 成员方法将序列化后的数据流图写入事件文件。除了数据流图外，表 6-1 列出了 FileWriter 写其他序列化数据的成员方法，它们都是继承自 SummaryToEventTransformer 的成员方法。它们的内部也都调用了 event_pb2.Event 方法将这些序列化数据转换成事件数据，然后调用 event_writer 的 add_event 方法将事件数据写入用户指定的事件文件中。

表 6-1　FileWriter 写序列化数据的成员方法

方法名称	功能说明
add_event	添加事件数据到事件文件
add_graph	添加数据流图到事件文件
add_meta_graph	添加元数据流图到事件文件
add_run_metadata	添加元数据信息到事件文件
add_session_log	添加会话日志到事件文件
add_summary	添加汇总数据到事件文件

add_* 方法并不会立即将数据写入事件文件。如果用户设置了 flush_secs 参数，FileWriter 便会定期将事件数据缓存队列中的数据写入事件文件。或者，用户还可以调用 flush 成员方法，手动刷新事件数据缓存队列。除此以外，表 6-2 还列出了 FileWriter 管理事件文件的其他成员方法。

表 6-2 FileWriter 管理事件文件的成员方法

方法名称	功能说明
get_logdir	获取事件文件的目录
flush	将事件数据缓存队列中的所有数据添加到事件文件中
close	先调用 flush 方法，然后关闭事件文件流
reopen	如果调用了 close 方法，则重新打开事件文件流，并在 logdir 目录下新建一个事件文件；否则，不做任何操作

6.3 可视化学习过程

使用 TensorFlow 训练深度神经网络模型的过程是复杂而漫长的，同时也像黑盒子一样充满了不确定性。为了清晰地理解深度神经网络模型的训练过程，同时方便地调试和优化训练程序，读者有必要掌握 TensorBoard 可视化学习的方法和对应的汇总操作。TensorBoard 支持的可视化对象主要包括模型参数和训练指标随时间的变化情况、特定值的数据分布和统计信息等。

6.3.1 汇总操作概述

汇总操作也是 TensorFlow 的操作，它的输入输出也是张量。但汇总操作与第 3 章中介绍的 3 类操作略有不同，因为它的输出是汇总数据。用户需要将这些汇总数据写入到事件文件中，TensorBoard 程序才能顺利加载并可视化它们。为了在会话运行过程中收集这些汇总数据，用户需要添加汇总操作，即在数据流图中添加对应的汇总节点。因此，我们认为汇总节点是有别于计算节点、存储节点和数据节点外的第四类节点，它的输出是特定的汇总数据。表 6-3 列出了 4 种典型汇总操作的功能说明。在会话中执行这些汇总操作，便可以获取一条汇总数据。执行频率越高，“采样率”越高。在资源充裕的情况下，甚至每一步训练都可以执行对应的汇总操作，但这可能导致资源开销倍增。

表 6-3 典型的 4 种汇总操作

操作名称	功能说明
tf.summary.scalar	获取一条带有标量值的汇总数据
tf.summary.histogram	获取一条带有统计值的汇总数据
tf.summary.image	获取一条带有图像的汇总数据
tf.summary.audio	获取一条带有音频的汇总数据

在表 6-3 列出的 4 种汇总数据中，标量值和统计值对应 summary.proto 文件定义的具名数值（Value）内嵌类，image 和 audio 则分别对应图像（Image）和音频（Audio）内嵌类。下面我们依次介绍这 4 种汇总操作输出的使用方法和对应汇总数据的可视化效果。

6.3.2　使用tf.summary.scalar生成折线图

在会话中执行 tf.summary.scalar 操作，可以获取一条带有标量值的汇总数据。表 6-4 列出了该操作的输入参数的功能说明。其中，collections 参数的默认值为 GraphKeys.SUMMARIES，因此所有的汇总操作默认都属于关键字为 GraphKeys.SUMMARIES 的操作集合。

表 6-4　tf.summary.scalar 的输入参数

参数名称	功能说明
name	生成的汇总节点名称，同时也是 TensorBoard 中显示的名称
tensor	对应标量值的张量
collections	汇总操作所属集合的关键字

下面是生成图 6-2 的部分代码：

```
with tf.name_scope('cross_entropy'):
  diff = tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y)
  with tf.name_scope('total'):
    cross_entropy = tf.reduce_mean(diff)
    # 添加获取交叉熵的汇总操作
    tf.summary.scalar('cross_entropy', cross_entropy)

with tf.name_scope('train'):
  train_step = tf.train.AdamOptimizer(0.001).minimize(
      cross_entropy)

with tf.name_scope('accuracy'):
  with tf.name_scope('correct_prediction'):
    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
  with tf.name_scope('accuracy'):
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    # 添加获取准确率的汇总操作
    tf.summary.scalar('accuracy', accuracy)
# 聚集汇总操作
merged = tf.summary.merge_all()
# 省略中间步骤
for i in range(FLAGS.max_step):
  if i % FLAGS.summary_step == 0:
    # 获取汇总数据和准确率
    summary, acc = sess.run([merged, accuracy], feed_dict=feed_dict(False))
    # 向事件文件中写入汇总数据，并传入全局步数
    writer.add_summary(summary, i)
```

结合上一节学习的名字作用域，我们重点关注 tf.summary.scalar 操作的用法。这里先后调用了两次 tf.summary.scalar 操作，分别添加了输出交叉熵和准确率的汇总操作。随后，我们调用 tf.summary.merge_all 方法添加了聚集汇总操作 merged。在会话中执行 merged 操作便可以获取所有的汇总数据，然后可以使用 FileWriter 将其写入事件文件中。

启动 TensorBoard 程序并读取上面代码生成的事件文件，便可以得到图 6-2 中的可视化效果。将其中的 accuracy 窗口展开，我们得到了图 6-8 所示的效果。下面依次介绍图 6-8 中 4 个方框中

各按钮和选项的功能。

首先，将鼠标指针移动到折线上（方框 1），可以得到对应时刻的汇总信息（黑色背景框），如光滑值、实际值、训练步数、时间戳等。TensorBoard 默认会用不同的颜色显示来自不同事件文件的数据。例如，图 6-8 分别显示了 MNIST softmax 模型在训练集和测试集上的结果。

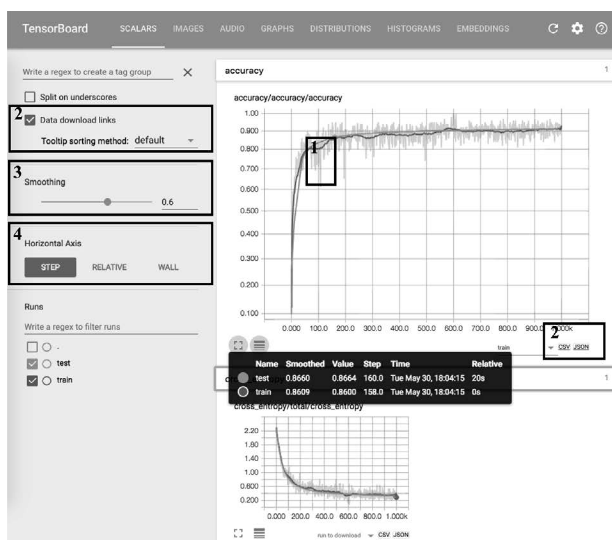


图 6-8 在 TensorBoard 中放大 accuracy 窗口的折线图

TensorBoard 支持下载任意指标随时间变化的结果（方框 2），目前提供 CSV 和 JSON 两种数据格式，并提供 3 种提示信息的排序方式：升序、降序或时间序。

图 6-8 中准确率折线图的光滑度为 0.6（方框 3）。折线图中前景层深色的线条为按照当前光滑值近似的曲线，背景层浅色的线条为实际值的折线。光滑度越低，曲线越接近真实值。图 6-9 展示了光滑度为 0.2 时的 accuracy 折线图。

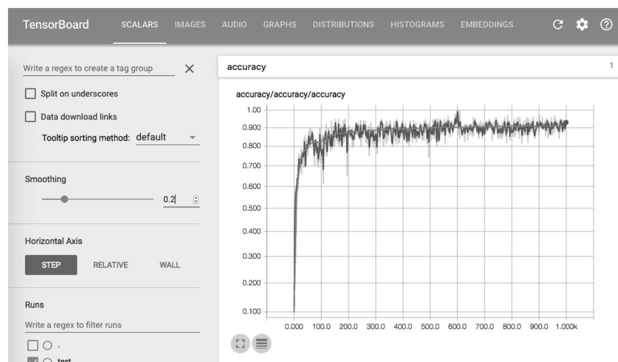


图 6-9 光滑度为 0.2 的 accuracy 折线图

所有折线图的横轴都提供 3 种显示模式（方框 4）：训练步数、相对时间和绝对时间。图 6-10 和图 6-11 分别给出了以相对时间和绝对时间为横轴的 accuracy 折线图。

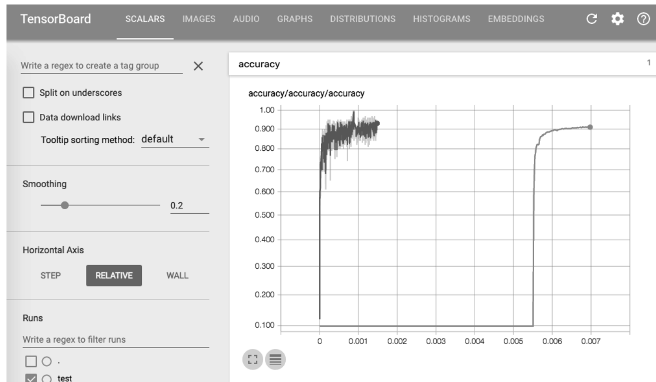


图 6-10　横轴为相对时间的 accuracy 折线图

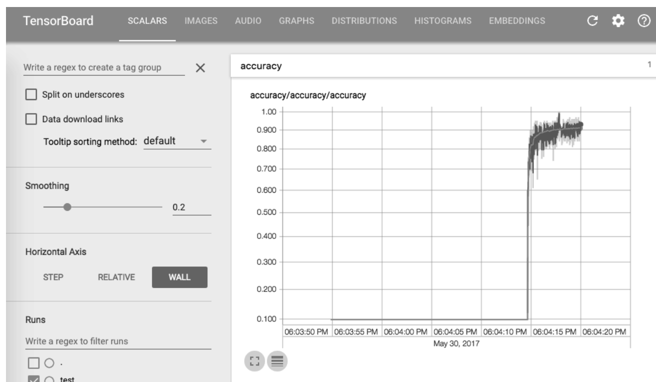


图 6-11　横轴为绝对时间的 accuracy 折线图

6.3.3　使用tf.summary.histogram生成数据分布图

在会话中执行 tf.summary.histogram 操作，可以获取一条带有统计值的汇总数据。它的输入参数与 tf.summary.scalar 操作类似，分别是 name、values 和 collections。其中，values 可以是任意形状的张量，其他两个输入参数与 tf.summary.scalar 操作没有区别。假设我们想要查看学习过程中模型权重值的数据分布和统计信息，修改后的部分代码如下所示：

```
with tf.name_scope('softmax_layer'):
  with tf.name_scope('weights'):
    weights = tf.Variable(tf.zeros([784, 10]))
    # 添加获取模型权重值的汇总操作
    tf.summary.histogram('weights', weights)
# 省略中间步骤
```

我们在会话中执行 `merged` 操作后，将获取的模型权重值的汇总数据写入事件文件，然后启动 TensorBoard 程序加载事件文件。最终结果如图 6-12 和图 6-13 所示。上面代码生成的模型权重的数据分布图和数据统计图分别展现在 DISTRIBUTIONS 和 HISTOGRAMS 面板下。

在图 6-12 所示的 DISTRIBUTIONS 面板中，上图和下图分别为测试集和训练集上模型权重的数据分布图。其中，竖轴是参数值，横轴是迭代步数，图中颜色深浅表示数据分布的稠密程度。一开始训练时，所有模型权重都被初始化为 0。随着训练步数的增加，权重矩阵中的值也开始趋于高斯分布。数据分布图默认只有图 6-12 下图的大小，我们需要点击方框才能将其放大到上图的大小。

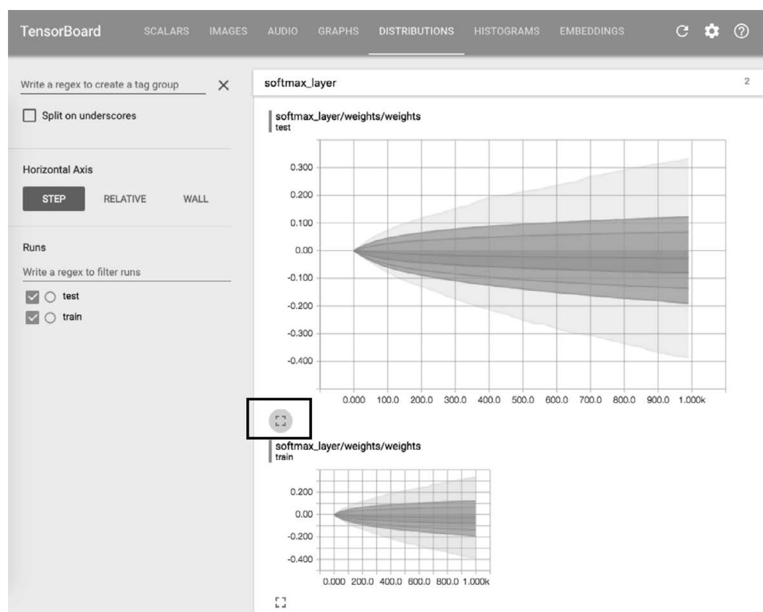


图 6-12 模型权重的数据分布图（另见彩插）

图 6-13 所示的 HISTOGRAMS 面板以另一种形式呈现模型权重的数据分布，它能够更清楚地体现出权重矩阵中所有元素值的统计信息。图 6-13 的上图和下图分别为测试集和训练集上模型权重的数据统计图。其中，竖轴是迭代步数。步数越大，颜色越浅、层次越靠前。横轴是权重值，每个取值都表示以该值为中心的一个区间。所有取值构成一个等差数列，区间长度等于等差数列的公差。例如，当横轴的取值为 [−3, −2, −1, 0, 1, 2, 3] 时，0 点对应的值表示权重值在 [−0.5, 0.5] 区间的元素个数。模型权重的实际分布决定了等差数列的公差。权重分布越稀疏，则公差越大；分布越稠密，则公差越小。在 HISTOGRAMS 面板中，我们还可以将鼠标指针移到山峰上，以便得到某个切面的详细信息。以图 6-13 为例，我们可以观察到，该算法模型在测试集上迭代到第 900 步时，以 0.0499 为中心值的区间内有 720 个元素。

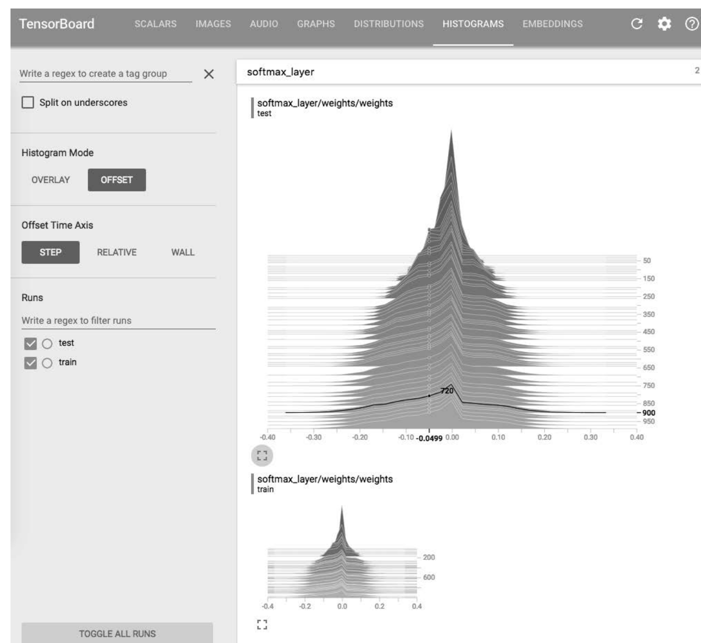


图 6-13　模型权重的数据统计图（另见彩插）

6.3.4　使用tf.summary.image生成图像

在会话中执行 `tf.summary.image` 操作，可以获取一条带有图像的汇总数据。它的输入参数如表 6-5 所示。`tensor` 参数必须是形如 `[batch_size, height, width, channels]` 的四阶张量，表示一批图像数据。`batch_size` 表示这批图像数据的个数；`height` 和 `width` 分别表示图像的高度和宽度，单位为像素；`channels` 表示图像的通道数，即色彩类型，可选值 1、3、4 分别表示灰度图（Grayscale）、彩色图（RGB）和带透明度的彩色图（RGBA）。`max_outputs` 参数表示在 TensorBoard 的 IMAGES 面板渲染图像的最大个数，它还决定了这些图像的标签名称后缀。如果 `max_outputs` 设为 1，则该图像的标签为 `{name}/image`；如果 `max_outputs` 大于 1，则这批图像的标签按顺序依次是 `{name}/image/0`、`{name}/image/1` 等。

表 6-5　`tf.summary.image` 的输入参数

参数名称	功能说明
`name`	生成的汇总节点名称，同时也是 TensorBoard 中显示的名称
`tensor`	表示图像数据的四阶张量
`max_outputs`	TensorBoard 渲染图像的最大个数
`collections`	汇总操作所属集合的关键字

我们现在希望可视化 MNIST 数据集中的手写体图像，那么需要将手写体图像数据转换为

`tf.summary.image` 操作接受的四阶张量。MNIST 数据集中的手写体图像数据是形状为 [1,784] 的张量，我们有必要将其转换成形如 [-1, 28, 28, 1] 的四阶张量。`batch_size` 设为–1 表示图像个数不定，在运行时根据形状推导赋值；`height` 和 `width` 均设为 28，尺寸与 MNIST 数据集中的手写体图像尺寸保持一致；`channels` 设为 1，表示该四阶张量需要转换成灰度图。然后，我们将该张量传入 `tf.summary.image` 操作，在会话中执行并获取到带有批图像的汇总数据，继而使用 `FileWriter` 将其写入事件文件。最后，使用 TensorBoard 程序加载事件文件，它会渲染出如图 6-14 所示的手写体图像。对应的代码片段如下所示：

```
with tf.name_scope('input'):
  x = tf.placeholder(tf.float32, [None, 784], name='x-input')
  y_ = tf.placeholder(tf.float32, [None, 10], name='y-input')

with tf.name_scope('input_reshape'):
  # 将输入图像 x 转换成四阶张量
  image_shaped_input = tf.reshape(x, [-1, 28, 28, 1])
  # 添加获取手写体图像的汇总操作，设置最大生成 10 张图像
  tf.summary.image('input', image_shaped_input, 10)
# 省略中间操作
```

因为 `max_outputs` 参数设置为 10，所以第 1 张图像的标签名称是 input_reshape/input/image/0。其中，`input_reshape` 是该汇总操作所属的作用域名称，input 是该图像的名称，0 表示它是这批图像的第 1 张。

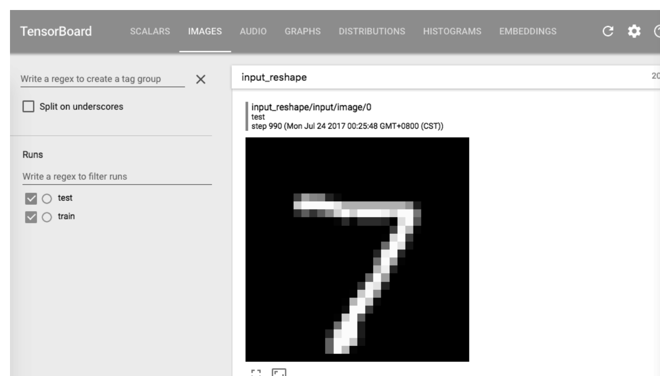


图 6-14　MNIST 数据集的手写体图像

6.3.5　使用tf.summary.audio生成音频

在会话中执行 `tf.summary.audio` 操作，可以获取一条带有音频的汇总数据。它的输入参数如表 6-6 所示。`tensor` 参数必须是形如 `[batch_size, frames, channels]` 的三阶张量或形如 `[batch_size, frames]` 的二阶张量，表示一批音频数据。`batch_size` 表示这批音频数据的个数；`frames` 表示音频帧的值，取值范围为 [–1.0, 1.0]。`sample_rate` 参数表示音频采样率。`max_outputs` 参数表示在 TensorBoard 的 AUDIO 面板下生成音频的最大个数，它同样也决定了

该音频标签名称的后缀。标签名称规则与 `tf.summary.image` 一样，这里不再赘述。

表 6-6　`tf.summary.audio` 的输入参数

参数名称	功能说明
`name`	生成的汇总节点名称，同时也是 TensorBoard 中显示的名称
`tensor`	表示音频数据的三阶或二阶张量
`sample_rate`	音频采样率
`max_outputs`	TensorBoard 生成音频的最大个数
`collections`	汇总操作所属集合的关键字

因为使用音频数据训练的模型还相对较少，加之不便于在书中展示，所以我们不再给出对应的代码片段。感兴趣的读者可以查看 Google Brain 的 magenta 项目，该项目旨在使用机器学习模型创造艺术和音乐作品。项目官网地址是 https://magenta.tensorflow.org。

6.3.6　可视化MNIST softmax模型学习过程的最佳实践

我们将本节介绍的汇总操作整合在一起，以 MNIST softmax 模型为例展示一个完整的可视化学习过程的最佳实践。下述代码生成的文件可以渲染出多种不同类型的学习过程图，具体包括准确率和交叉熵的折线图、模型参数的数据分布图和数据统计图，以及输入手写体数字图像和 MNIST softmax 模型的数据流图：

```
# 6.4_best_practice.py
# -*- coding:utf-8 -*-
import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input_data

# 导入 MNIST 数据集
mnist = input_data.read_data_sets('/tmp/tensorflow/mnist/input_data', one_hot=True)

with tf.name_scope('input'):
  x = tf.placeholder(tf.float32, [None, 784], name='x-input')
  y_ = tf.placeholder(tf.float32, [None, 10], name='y-input')

with tf.name_scope('input_reshape'):
  # 将输入图像 x 转换成四阶张量
  image_shaped_input = tf.reshape(x, [-1, 28, 28, 1])
  # 添加获取手写体图像的汇总操作，设置最大生成 10 张图像
  tf.summary.image('input', image_shaped_input, 10)

with tf.name_scope('softmax_layer'):
  with tf.name_scope('weights'):
    weights = tf.Variable(tf.zeros([784, 10]))
    # 添加获取模型权重值的汇总操作
    tf.summary.histogram('weights', weights)
  with tf.name_scope('biases'):
    biases = tf.Variable(tf.zeros([10]))
```

```
    # 添加获取模型偏置值的汇总操作
    tf.summary.histogram('biases', biases)
  with tf.name_scope('Wx_plus_b'):
    y = tf.matmul(x, weights) + biases

with tf.name_scope('cross_entropy'):
  diff = tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y)
  with tf.name_scope('total'):
    cross_entropy = tf.reduce_mean(diff)
    # 添加获取交叉熵的汇总操作
    tf.summary.scalar('cross_entropy', cross_entropy)

with tf.name_scope('train'):
  train_step = tf.train.AdamOptimizer(0.001).minimize(
      cross_entropy)

with tf.name_scope('accuracy'):
  with tf.name_scope('correct_prediction'):
    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
  with tf.name_scope('accuracy'):
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    # 添加获取准确率的汇总操作
    tf.summary.scalar('accuracy', accuracy)

merged = tf.summary.merge_all()
sess = tf.InteractiveSession()

train_writer = tf.summary.FileWriter('/tmp/summary/mnist' + '/train', sess.graph)
test_writer = tf.summary.FileWriter('/tmp/summary/mnist' + '/test')
tf.global_variables_initializer().run()

def feed_dict(train):
  """填充训练数据或测试数据的方法"""
  if train:
    xs, ys = mnist.train.next_batch(100, fake_data=False)
  else:
    xs, ys = mnist.test.images, mnist.test.labels
  return {x: xs, y_: ys}

for i in range(1000):
  if i % 10 == 0:  # 写汇总数据和测试集的准确率
    summary, acc = sess.run([merged, accuracy], feed_dict=feed_dict(False))
    test_writer.add_summary(summary, i)
    print('Accuracy at step %s: %s' % (i, acc))
  else:  # Record train set summaries, and train
    if i % 100 == 99:  # 写运行时的事件数据
      run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
      run_metadata = tf.RunMetadata()
      summary, _ = sess.run([merged, train_step],
                            feed_dict=feed_dict(True),
                            options=run_options,
                            run_metadata=run_metadata)
      train_writer.add_run_metadata(run_metadata, 'step%03d' % i)
      train_writer.add_summary(summary, i)
```

```
        print('Adding run metadata for', i)
      else:  # 写汇总数据
        summary, _ = sess.run([merged, train_step], feed_dict=feed_dict(True))
        train_writer.add_summary(summary, i)

  train_writer.close()
  test_writer.close()
```

6.4　可视化高维数据

本节讨论的可视化高维数据特指嵌入表示的高维向量。嵌入是指将客观世界中离散的物体或对象（如单词、短语、图片等）映射到特征空间中的操作，嵌入向量是指映射后的特征空间中连续且稠密的高维向量。因为深度学习和机器学习的大多数模型都是为稠密的连续向量而设计的，所以嵌入表示使得我们可以使用经典的算法模型处理许多离散的输入向量。

在语音识别、图像识别和推荐系统等经典的深度学习和机器学习应用场景中，我们通常使用嵌入向量来描述客观世界的物体。嵌入向量不是对物体进行简单编号的结果，而是在尽量保持相似性不变的前提下对物体进行特征抽象和编码的产物。通过不断训练，我们能够将客观世界中的物体“不失真”地映射到高维特征空间中，进而可以使用这些嵌入向量实现分类、回归和预测等操作。为了能够一窥神经网络黑盒中的真相，有必要学会使用 TensorBoard 提供的高维数据可视化能力。本节首先介绍 TensorBoard 可视化高维数据的推荐流程和方法，然后介绍使用 TensorBoard 可视化 MNIST 数据集的最佳实践。

6.4.1　使用TensorBoard可视化高维数据

TensorBoard 可视化高维数据的模块叫作嵌入投影仪（embedding projector），打开 EMBEDDINGS 面板即可查看。嵌入投影仪是将高维数据投影到低维空间（如二维和三维）中的可视化模块，它内置 t-SNE（t-Distributed Stochastic Neighbor Embedding）和主成分分析（Principal Component Analysis，PCA）这两种降维算法，也支持用户自定义降维算法。因为嵌入投影仪是 TensorBoard 集成的第三方组件，所以它与原生可视化功能的使用方法略有不同。如图 6-15 所示，TensorBoard 可视化高维数据的核心在于 3 类对象及其对应的 3 种文件的使用。3 类对象分别是嵌入变量、嵌入变量元数据和投影配置参数。保存它们的 3 种文件分别是 checkpoint 文件、元数据文件和投影配置文件。下面我们分别介绍这 3 组对象和文件的使用方法。

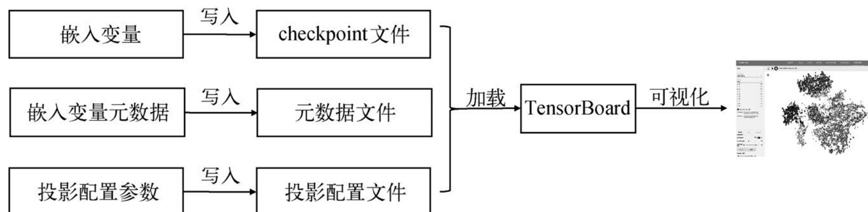


图 6-15　使用 TensorBoard 可视化高维数据的典型流程

- **嵌入变量**：它是待可视化数据的载体。TensorBoard 要求用户使用形如[嵌入数据总数,嵌入数据维度]的二阶张量表示想要可视化的高维数据。嵌入变量本身也是一种变量，因此我们通常将其命名为 embedding_var，并将其写入 checkpoint 文件。下面的示例代码展示了如何创建一个保存 10000 幅 MNIST 手写体数字图像的嵌入变量：

```
# 创建嵌入变量，保存测试集中的 10000 张手写体数字图像
embedding_var = tf.Variable(tf.stack(mnist.test.images[:10000]),
                            trainable=False, name='embedding')
# 创建 Saver 对象，并保存嵌入变量
saver = tf.train.Saver()
saver.save(sess, os.path.join(FLAGS.log_dir + '/model.ckpt'))
```

- **嵌入变量元数据**：用于描述嵌入变量中每个元素的数据特征集合。在 TensorBoard 中展示投影的数据点时，我们可以为其添加对应的数据特征，以便更清晰地观察数据。比如，我们可以为 MNIST 图像的嵌入变量添加对应的数字标签，或者为词向量的嵌入变量添加对应的出现频数。TensorFlow 推荐使用 TSV 文件格式保存嵌入变量元数据。该格式类似于 CSV 格式，第一行为数据头，第二行开始为真正的数据。如果嵌入变量包含多个不同的数据特征，那么特征间使用 tab 分隔。因为元数据在训练过程中不会更新，所以将其独立保存在 metadata.tsv 文件中有利于 TensorBoard 快速加载。在为 MNIST 数据集添加数字标签的例子中，因为只有标签这一个数据特征，所以可以省略数据头，直接将数据特征写入元数据文件。下面给出 MNIST 数据集的元数据文件示例：

```
7
2
1
...
```

对于表示图像的嵌入变量，TensorBoard 还支持为其添加对应的图像文件，以供页面呈现。TensorFlow 官方提供了包含 MNIST 测试集中全部 10000 张手写体数字的全景图（https://www.tensorflow.org/images/mnist_10k_sprite.png）。在计算机图形学领域，这种由多幅子图集成的全景图也称为 sprite image。只要保证全景图中的手写体数字顺序与 metadata.tsv 中的顺序一致，TensorBoard 便能够正确找到数字对应的图片。TensorBoard 切分全景图的默认顺序是先从左到右，再从上到下。只要我们明确设置了每张手写体图像的尺寸，TensorBoard 便可以从全景图中按序裁剪出每一幅手写体图像，在页面上独立渲染。图 6-16 展示了这张全景图的开头部分。

图 6-16 MNIST 数据集中的部分手写体数字的图像

- **投影配置参数**：用于指定TensorBoard可视化高维数据时所使用的嵌入变量及其元数据。投影配置参数存储在以Protocol Buffer序列化格式保存的文本文件中，这些配置主要包括嵌入变量名称（tensor_name）、元数据文件路径（metadata_path）和全景图配置（sprite）等。通常，我们将投影配置文件命名为 projector_config.pbtxt。下面我们仍然以可视化MNIST数据集为例说明。假设之前定义的嵌入变量embedding_var在数据流图中的名称为 embedding:0，元数据文件路径为 /tmp/summary/embeddings/metadata.tsv，MNIST 测试集的全景图文件路径为 /tmp/summary/images/mnist_10k_sprite.png，且每一幅手写体图像的常和宽均为 28 像素，那么投影配置文件的定义如下所示：

```
embeddings {
  tensor_name: "embedding:0"
  metadata_path: "/tmp/summary/embeddings/metadata.tsv"
  sprite {
  image_path: "/tmp/summary/images/mnist_10k_sprite.png"
  single_image_dim: 28
  single_image_dim: 28
  }
}
```

总结 TensorBoard 可视化高维数据的流程，我们认为它可以分为以下 4 步。

(1) 创建嵌入变量，并保存到 checkpoint 文件。

(2) 创建嵌入变量元数据，并保存到元数据文件（通常命名为 metadata.tsv）。

(3) 设置投影配置参数，保存到投影配置文件（通常命名为 projector_config.pbtxt）。

(4) 运行 TensorBoard 程序，加载上面生成的 3 个文件，可视化高维数据的投影结果。

6.4.2　可视化MNIST数据集的最佳实践

无论是展示图像识别模型的训练效果，还是展示高维数据的可视化效果，MNIST 数据集一直以来都是人们的优先选择。一来是因为手写体数字识别的技术已经非常成熟，基于卷积神经网络模型的准确率可以达到 99% 以上；二来是因为从 0 到 9 这 10 个数字本身特征明显，聚类后的可视化效果比较直观。下面的代码创建并保存了可视化 MNIST 数据集所需的 checkpoint 文件、元数据文件和投影配置文件：

```
# 6.4_best_practice.py
# -*- coding:utf-8 -*-
import argparse
import sys
import os

import numpy as np
import tensorflow as tf

from tensorflow.contrib.tensorboard.plugins import projector
from tensorflow.examples.tutorials.mnist import input_data
```

```
FLAGS = None

def main(_):
  # 创建日志目录
  if tf.gfile.Exists(FLAGS.log_dir):
      tf.gfile.DeleteRecursively(FLAGS.log_dir)
  tf.gfile.MakeDirs(FLAGS.log_dir)
  # 读取 MNIST 数据集
  mnist = input_data.read_data_sets(FLAGS.data_dir,
                                    one_hot=True,
                                    fake_data=FLAGS.fake_data)
  # 创建嵌入变量，保存测试集中的 10000 张手写体数字图像
  embedding_var = tf.Variable(tf.stack(mnist.test.images[:10000]),
                              trainable=False, name='embedding')
  # 创建交互式会话，并初始化全局变量
  sess = tf.InteractiveSession()
  tf.global_variables_initializer().run()
  # 创建 Saver 对象，将嵌入变量保存到 checkpoint 文件中
  saver = tf.train.Saver()
  saver.save(sess, os.path.join(FLAGS.log_dir + '/model.ckpt'))
  # 创建元数据文件，并将手写体数字对应的标签写入元数据文件
  metadata_file = FLAGS.log_dir + '/metadata.tsv'
  with open(metadata_file, 'w') as f:
    for i in range(FLAGS.max_nums):
      c = np.nonzero(mnist.test.labels[::1])[1:][0][i]
      f.write('{}\n'.format(c))
  # 创建 FileWriter，并保存数据流图
  writer = tf.summary.FileWriter(FLAGS.log_dir, sess.graph)
  # 创建投影配置参数
  config = projector.ProjectorConfig()
  embeddings= config.embeddings.add()
  embeddings.tensor_name = 'embedding:0'
  embeddings.metadata_path = os.path.join(FLAGS.log_dir + '/metadata.tsv')
  # 设置全景图文件路径和手写体数字图像的尺寸
  embeddings.sprite.image_path = os.path.join('/tmp/summary/images/mnist_10k_sprite.png')
  embeddings.sprite.single_image_dim.extend([28, 28])
  # 执行 visualize_embeddings 方法，将参数配置写入新创建的投影配置文件中
  # TensorBoard 启动时会自动加载该文件中的投影参数配置
  projector.visualize_embeddings(writer, config)

if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--fake_data', nargs='?', const=True, type=bool,
                      default=False,
                      help='If true, uses fake data for unit testing.')
  parser.add_argument('--max_nums', type=int, default=10000,
                      help='Number of steps to run trainer.')
  parser.add_argument('--data_dir', type=str,
                      default='/tmp/tensorflow/mnist/input_data',
                      help='Directory for storing input data')
  parser.add_argument('--log_dir', type=str,
                      default='/tmp/summary/embeddings',
                      help='Summaries log directory')
```

```
    FLAGS, unparsed = parser.parse_known_args()
    tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

执行上面的代码时，读者应根据本地环境设置对应的输入参数。其中，`data_dir` 参数表示读取的 MNIST 数据集的目录，`log_dir` 表示生成日志文件的目录。程序执行后，生成的文件目录结构如下所示：

```
/tmp/summary
|-- embeddings
|   |-- checkpoint
|   |-- events.out.tfevents.1501427485.Django.local
|   |-- metadata.tsv
|   |-- model.ckpt.data-00000-of-00001
|   |-- model.ckpt.index
|   |-- model.ckpt.meta
|   `-- projector_config.pbtxt
|-- images
|   `-- mnist_10k_sprite.png
```

然后，启动 TensorBoard 程序，并设置 `logdir` 参数为 /tmp/summary/embeddings，相关命令如下：

```
$ tesorboard --logdir=/tmp/summary/embeddings
```

最后，在浏览器的地址栏中输入 TensorBoard 的 URL，并进入 EMBEDDINGS 面板。TensorBoard 默认使用 PCA 降维，我们需要手动选择 t-SNE 降维方法（方框 1），并设置使用不同颜色区分不同标签（方框 2）。经过多轮迭代后，我们最终得到了如图 6-17 所示的分类效果。由于我们设置了全景图元信息，这里展示的每张手写体数字都是从全景图中提取的原始图像。

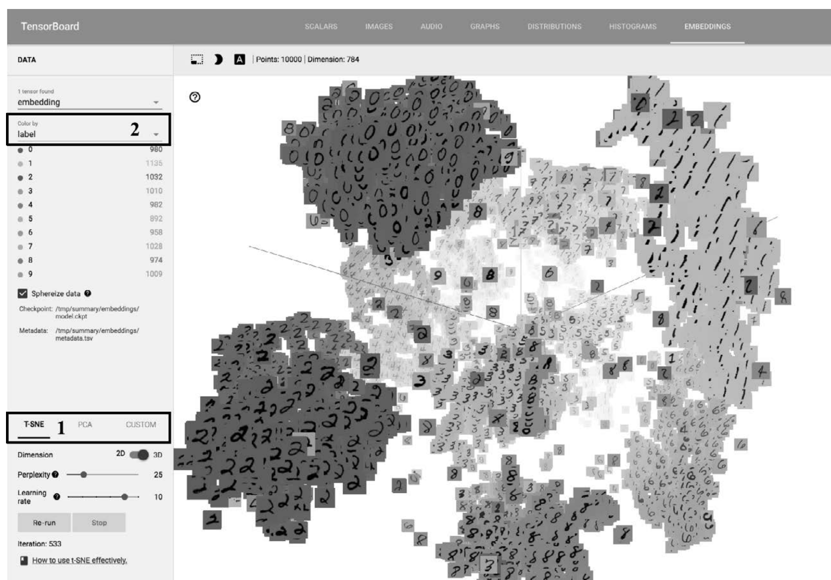


图 6-17　使用 TensorBoard 可视化 MNIST 数据集（另见彩插）

6.5　小结

TensorBoard 是一款优秀的深度学习可视化工具，它能够通过 Web 界面直观地展示 TensorFlow 模型开发和训练过程中诸多类型的抽象数据，并提供一定的动态特征和交互能力。在 TensorBoard 的核心功能中，可视化数据流图主要用于调测模型结构，可视化度量指标和模型参数主要用于评估模型学习效果，可视化图像和音频数据主要用于观察模型输入输出，可视化高维数据主要用于展示高维嵌入数据的分布情况。使用 TensorBoard 时，用户需要在程序中引入 tf.summary 模块提供的工具，输出必要的序列化数据。希望读者在今后的模型开发和使用中，能够灵活运用 TensorBoard 提供的可视化能力，提升工作效率。

第 7 章

模型托管工具：TensorFlow Serving

基于 TensorFlow 提供的基础抽象和编程框架开发算法模型仅仅是构建实用化深度学习系统的第一步。在应用系统中，模型通常会以推理（预测）态的形式工作，服务于软件的整体流程。TensorFlow Serving 是一套标准化的模型托管工具，它能够打通从模型训练到发布的全流程，并以较低的管理成本和便捷的部署方式为用户提供在线推理服务。本章首先对 TensorFlow Serving 进行整体性说明，给出其工作流水线。继而阐述它的系统架构和核心概念，以及各个功能模块的组织结构和调用关系。最后从实践入手，介绍 TensorFlow Serving 运行环境的搭建和模型托管的最佳实践。

7.1 概述

对于传统软件，我们已经拥有大量实现持续集成和迭代发布的 DevOps 工具。然而，在深度学习和机器学习模型的持续集成方面，业界目前还处于起步阶段。模型训练同软件开发相比，具有一定差异性。软件开发需要不断更新代码，然后执行单元测试和集成测试，进而迭代发布版本；而模型训练几乎不更新代码，仅需不断输入新的批数据，从而不断训练模型并更新模型参数即可。为了帮助用户将持续迭代训练的模型快速集成到线上系统或发布为服务，Google 公司研发了一套兼具灵活性和高性能的模型托管工具——TensorFlow Serving。

TensorFlow Serving 支持生产级的服务部署，允许用户快速搭建从模型训练到服务发布的工作流水线。它支持将多个模型组合为一个完备服务进行发布，也能够管理一个模型服务的多个版本。TensorFlow Serving 可以在保持服务架构和 API 不变的情况下，更新线上模型和运行环境，使得模型的在线学习和增量学习成为可能。TensorFlow Serving 不但能够实现 TensorFlow 模型的开箱即用集成，而且提供对自定义模型的可扩展能力。通过编写简单的插件，用户即可使用 TensorFlow Serving 托管自定义类型的模型。

图 7-1 展示了 TensorFlow Serving 的工作流水线，这一流水线主要由以下 3 部分组成。

- **持续训练过程**：基于持续输入的批数据，使用 TensorFlow 不断训练模型，并将模型定期保存到指定目录。
- **模型服务**：发布训练好的模型，对外提供基于 gRPC 协议的模型服务 API，同时支持模型的不断更新。
- **客户端访问**：使用基于 gRPC 协议的客户端程序向模型服务发起请求，并等待服务的响应结果。

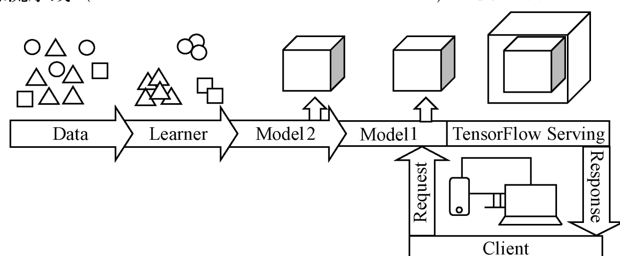


图 7-1 TensorFlow Serving 的工作流水线

7.2 系统架构

TensorFlow Serving 采用经典的客户端–服务端架构提供模型服务。服务端主要由 TensorFlow Serving 核心类和接口，以及模型相关的插件模块组成。客户端是指用户面向模型推理等业务需求，设计实现的 TensorFlow Serving 服务请求代码。

图 7-2 展示了 TensorFlow Serving 的系统架构。其中，`Loader`、`Source`、`Manager` 和 `ServableHandle` 是基于 C++ 实现的核心类和接口。`Servable`、`SavedModel` 和 `VersionPolicy` 是模型相关的插件模块。Client 是用户实现的 gRPC 客户端，用于向 `Manager` 加载模型服务发起请求。下面我们依次介绍图中的各个概念，帮助读者快速理解 TensorFlow Serving 的系统架构和工作原理。

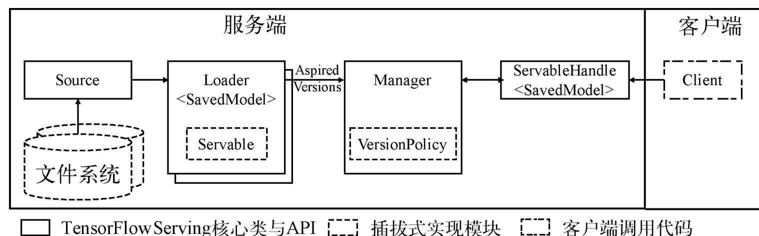


图 7-2 TensorFlow Serving 系统架构

`Servable` 是 TensorFlow Serving 服务端提供计算和查询等服务的实例，它是 TensorFlow Serving 系统中最核心的概念。`Servable` 兼具通用性和灵活性的特点。一套 TensorFlow Serving 系统支持同时运行多个服务。为了区分不同的服务，我们需要为每个 `Servable` 设置不同的名称。

服务也具有版本的概念。以模型服务为例，我们在发布服务后通常还会不断训练和保存模型，因此总是存在使用新模型替换线上旧模型的情况。这说明运行中的服务在不同时刻加载的可能是同一个模型的不同版本。为了便于服务管理和版本更新，TensorFlow Serving 将每个已发布服务的后续更新版本都统一保存在各自的 Servable Stream 队列中。每个 `Servable` 都拥有一个唯一的 Servable Stream 队列。队列中的元素按照版本号大小升序排列，并共享同一个 `Servable` 名称。

`Loader` 是 `Servable` 的加载器抽象，它提供了一套加载和卸载 `Servable`，以及评估系统资源是否足够加载 `Servable` 的标准编程接口。`Loader` API 具有良好的通用性设计，它独立于特定的服务、算法、数据和产品用例。每个 `Loader` 都只负责一个对应 Servable Stream 的加载、卸载和资源评估工作。同一时刻，每个 `Loader` 只能加载对应 Servable Stream 中一个特定版本的 Servable。TensorFlow Serving 将 Servable Stream 中这些等待被加载的 Servable 集合定义为 Aspired Versions。

`SavedModel` 是 TensorFlow 模型持久化存储的通用序列化格式，也是打通从模型训练到服务发布流程的关键。在模型训练阶段，我们有可能会使用输入流水线和超参数优化操作（如 Dropout）。但是，当我们准备将模型保存并发布为服务时，往往需要部分删除或者替换这些操作，否则会出现推理（预测）准确率较低，甚至输入队列阻塞的情况。为了尽可能提升服务发布后的准确率或其他评价指标，我们需要保存多份不同的数据流图进行测试。`SavedModel` 为解决这个问题提供了便利：它支持使用同一份 saved_model.pb 文件来保存多幅不同的数据流图，这些数据流图可以共享模型参数和资源。这允许我们测试同一模型的多种不同操作组合，以便快速确定最优的数据流图。

`Source` 是数据处理器抽象，它负责监控和处理 `Servable` 加载的数据，比如文件系统指定路径下的查找表（lookup table）文件或模型文件。当 `Source` 观察到指定路径下有新版本的模型文件生成，或既有模型文件中的参数更新时，它便会向对应的 Servable Stream 中添加新版本的 `Servable`，并向服务管理器发起一个加载该 `Servable` 的请求。一个 `Source` 可以同时维护多个不同的 Servable Stream。

`Manager` 是服务的管理器抽象。它根据可用资源和版本更新策略决定是否允许 `Loader` 加载或卸载 Servable Stream 中的 `Servable`。`Manager` 提供如下 3 种获取指定 `Servable` 的方法：

- ServableStream 中指定版本的 `Servable`；
- ServableStream 中最新版本的 `Servable`；
- 指定 `ServableId` 的 `Servable`。

`VersionPolicy` 是 `Manager` 中的服务版本更新策略，可由用户根据服务需求自定义。不同 `VersionPolicy` 的侧重点各不相同，典型的策略有以下两种。

- **资源节省性策略**。该策略是 TensorFlow Serving 内置的默认策略，它要求 `Manager` 先卸载旧版本的 `Servable` 再加载新版本的 `Servable`。它优先减少服务的资源开销，然而服务更新时会有短暂的中断。

- **服务稳定性策略**。该策略要求 `Manager` 先加载新版本的 `Servable`，并开始响应服务；等待旧版本 `Servable` 之前接收的请求处理完毕后，再将其卸载。它优先保证服务的可用性，缺点是资源消耗更大。

`ServableHandle` 是响应 gRPC 客户端访问请求的服务句柄。`ServableHandle` 与已加载的 `Servable` 一一对应。客户端请求 TensorFlow Serving 发布的服务时，服务端的 `ServableHandle` 会调用相应 `Servable` 的服务接口，并将其响应返回给客户端。如果服务尚未加载，则向客户端返回错误信息。

以上介绍了 TensorFlow Serving 服务端的逻辑抽象。它们规范了 TensorFlow 模型保存和导出的标准格式，定义了加载和卸载模型服务的标准接口。在物理实现时，TensorFlow Serving 项目编译后生成的可执行模块是 `ModelServer`。`ModelServer` 是 TensorFlow Serving 系统的运行时主体。它允许用户加载文件系统指定目录下的模型文件或查找表文件，并将其发布到本地网络端口以提供模型服务或其他通用服务。用户可以通过 gRPC 协议，按照指定的请求格式调用服务。为了方便部署，大多数用户通常选择在物理服务器上直接使用 `ModelServer` 托管服务。不过，为了提升 `ModelServer` 应用程序本身的可移植性和自愈性，我们还可以将 `ModelServer` 运行在 Docker 容器或者 Kubernetes 集群中。

图 7-3 展示了 TensorFlow Serving 的运行时组件交互关系。其中客户端发送的输入数据和服务端返回的输出数据均由 Protocol Buffers 定义，客户端与服务端通过 gRPC 协议交互。客户端首先将输入数据序列化为 Protocol Buffers 数据类型，接着调用服务端提供的 gRPC 接口发起服务请求；服务端首先将接收到的 Protocol Buffers 数据类型反序列化为输入数据，接着调用 `ModelServer` 加载的对应服务，并得到服务输出的结果，最后将其序列化为 Protocol Buffers 数据类型并返回给客户端。

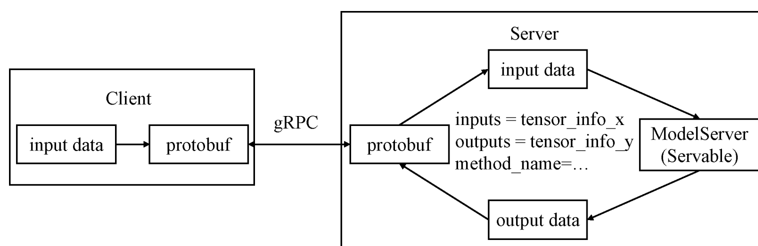


图 7-3 TensorFlow Serving 的运行时组件交互关系

7.3 安装

TensorFlow Serving 的安装主要是指其服务端核心组件——ModelServer 的安装，同时也涉及若干外部依赖项和周边组件。ModelServer 支持二进制包和源代码编译两种安装方式。

7.3.1　使用APT安装ModelServer

TensorFlow 官方目前支持使用 APT（Advanced Packaging Tool，高级包管理工具）在 Debian 及其派生的 Linux 发行版（如 Ubuntu）上安装 ModelServer 的二进制包。ModelServer 二进制包根据编译时的优化选项不同，分为以下两种。

- tensorflow-model-server：全面优化的 ModelServer，它使用针对特定操作系统和硬件平台优化的编译选项，比如开启了 SSE4 和 AVX 指令集支持。因此，可能出现无法在老的操作系统和硬件平台上顺利运行的情况。我们建议用户优先安装和测试 tensorflow-model-server。
- tensorflow-model-server-universal：更通用的 ModelServer，没有使用针对特定操作系统和硬件平台的编译优化选项。因此，它可以在绝大多数环境中顺利运行。当 tensorflow-model-server 无法正常使用时，用户可以选择它作为替代品。

我们推荐用户按照以下步骤安装 ModelServer 的二进制包。

首先，为 APT 添加 TensorFlow Serving 的资源下载路径。然后，将 Google 发布的 TensorFlow Serving GPG 公钥添加到 APT 的密钥库。相关代码如下：

```
$ echo "deb [arch=amd64] http://storage.googleapis.com/tensorflow-serving-apt stable
tensorflow-model-server tensorflow-model-server-universal" | sudo tee
/etc/apt/sources.list.d/tensorflow-serving.list
$ curl
https://storage.googleapis.com/tensorflow-serving-apt/tensorflow-serving.release.pub.gpg |
sudo apt-key add -
```

接着，更新 APT 的资源列表，并安装 tensorflow-model-server：

```
$ sudo apt-get update
$ sudo aptget install tensorflow-model-server
Preparing to unpack .../tensorflow-model-server_1.3.0_all.deb ...
Unpacking tensorflow-model-server (1.3.0) ...
Setting up tensorflow-model-server (1.3.0) ...
```

安装完成后，我们可以在命令行界面下检查是否能够调用 `tensorflow_model_server` 可执行文件。如果 `tensorflow_model_server` 打印出如下所示的 ModelServer 使用方法，那么说明安装成功：

```
$ tensorflow_model_server
usage: tensorflow_model_server
Flags:
        --port=8500                             int32   port to listen on
        --enable_batching=false                 bool    enable batching
...
```

如果操作系统和硬件平台不支持某些高级编译选项，或因为其他原因导致 tensorflow-model-server 不能正常工作，那么应该安装更通用的 tensorflow-model-server-universal。在安装通用版本之前，我们应该先卸载已经安装的优化版本，以避免因文件冲突而产生错误。

7.3.2 使用源码编译安装ModelServer

如果用户使用 macOS 或 Red Hat Enterprise Linux 等不依赖于 APT 管理软件包的操作系统，那么就需要从 TensorFlow Serving 源码编译安装 ModelServer。源码构建 ModelServer 的依赖项包括 Bazel、gRPC 和若干系统软件包。下面我们介绍使用 TensorFlow Serving 源码编译安装 ModelServer 的推荐步骤。

(1) 安装 Bazel 软件构建工具，具体可参考 2.1.6 节。

(2) 安装 gPRC 远程过程调用框架的 Python 工具包：

```
$ pip install grpcio
```

(3) 安装 TensorFlow Serving 所依赖的系统软件包。下面列出了在 Ubuntu 14.04 系统上构建 TensorFlow Serving 所需的系统软件包：

```
build-essential
curl
libcurl3-dev
git
libfreetype6-dev
libpng12-dev
libzmq3-dev
pkg-config
python-dev
python-numpy
python-pip
software-properties-common
swig
zip
zlib1g-dev
```

其他操作系统对应的软件包名称可能略有不同。用户可查阅相关软件文档，安装对应当前系统的等价软件包。

(4)（可选步骤）安装 TensorFlow Serving Python API 软件包。客户端需要调用 Protocol Buffers 定义的 TensorFlow Serving API。当使用 Python 语言开发客户端时，如果用户不想通过 .proto 文件自行构建 Python API，则可以直接安装预编译好的 tensorflow-serving-api 软件包：

```
$ pip install tensorflow-serving-api
```

(5) 完整克隆 TensorFlow Serving 项目及其子模块的代码：

```
$ git clone --recurse-submodules https://github.com/tensorflow/serving
```

(6) 运行 configure 脚本，配置 TensorFlow 编译选项，具体可参考 2.1.6 节：

```
$ cd serving/tensorflow
$ ./configure
```

(7) 编译 TensorFlow Serving 源码。编译完成后，可以在 tensorflow_serving/model_servers 目录下找到 ModelServer 的可执行文件——`tensorflow_model_server`：

```
$ cd ..
$ bazel build -c opt tensorflow_serving/...
$ bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server
usage: tensorflow_model_server
...
```

(8) 测试 ModelServer 是否正确编译。我们可以使用 `bazel test` 命令运行 TensorFlow Serving 提供的测试用例。下面的命令行输出显示所有测试用例通过，说明源码编译的 ModelServer 工作正常：

```
$ bazel test -c opt tensorflow_serving/...
...
Executed 58 out of 58 tests: 58 tests pass.
```

需要说明的是，在步骤 (7) 中，我们没有使用附加的编译选项，这可以确保软件适用于绝大多数运行环境。我们也可以根据软硬件环境的实际情况开启下列编译优化特性，它们将有效提升 TensorFlow 和 ModelServer 的计算性能：

```
bazel build -c opt --config=mkl --copt=-msse4.1 --copt=-msse4.2 --copt=-mavx --copt=-mavx2
--copt=-mfma --copt=-O3 tensorflow_serving/...
```

7.4　最佳实践

TensorFlow Serving 的托管模型流程主要包含 4 个步骤，分别是训练模型、导出模型、发布模型服务和更新线上模型服务。其中，训练模型的方法与前面章节介绍的原生 TensorFlow 模型训练过程并无差异。为了使读者快速上手，本节以 MNIST 模型为例，基于 TensorFlow Serving 项目提供的客户端，完整地展示使用 TensorFlow Serving 托管模型的全流程。

7.4.1　导出模型

TensorFlow Serving 的服务端组件只负责模型服务的发布和更新，模型的导出需要由 TensorFlow 原生的 `tensorflow.saved_model.builder.SavedModelBuilder` 模块实现。按照用户设置的模型导出目录和周期，`SavedModelBuilder` 定期向文件系统中的目标路径导出模型快照。模型快照的格式为 7.2 节介绍的 `SavedModel` 序列化格式，不同时刻导出的模型会被标记为不同的版本。下面是 SavedModelBuilder 导出的模型目录的典型结构示例：

```
├── 1
│   ├── saved_model.pb
│   └── variables
│       ├── variables.data-00000-of-00001
│       └── variables.index
```

目录名称 1 表示该目录保存的模型版本号为 1。在该目录中，save_model.pb 文件是序列化后的 `tensorflow::SavedModel` 对象，它包括模型对应的一幅或多幅数据流图，以及模型的元数据。variables 子目录下的 variables.data-*xxxxx*-of-*xxxxx* 文件是序列化后的不同版本的模型参数；variables. index 文件保存着模型参数名称和图中节点的索引，便于快速恢复模型参数。

我们在第 4 章中介绍过使用 Saver 保存模型的方法，也在上一章介绍了使用 FileWriter 保存数据流图和汇总数据的方法。SavedModelBuilder 的使用方法与前两者略有不同。由于 SavedModelBuilder 导出的模型会以服务的形式对外开放，我们需要在导出模型时明确定义请求服务的接口。TensorFlow Serving 推荐使用 SignatureDef Protocol Buffers 来定义接口，接口定义具体包括输入参数、输出参数和服务响应方法的名称。SignatureDef 对应的工具类提供了构建 SignatureDef Protocol Buffers 的方法——tf.saved_model.signature_def_utils.build_signature_def。

使用 SavedModelBuilder 导出模型分为以下 3 个步骤。

(1) 构造 SavedModelBuilder 实例，并设置模型的导出路径。

(2) 定义模型服务的 SignatureDef。

(3) 使用 SavedModelBuilder 实例导出模型。

为了方便说明，我们以 TensorFlow Serving 项目的示例代码 tensorflow_serving/example/mnist_saved_model.py 为例，介绍导出模型的具体实现。下面是我们从 mnist_saved_model.py 文件中摘取的主要代码片段：

7

```
import tensorflow as tf
# 训练 MNIST softmax 模型
export_path_base = sys.argv[-1]
export_path = os.path.join(
      compat.as_bytes(export_path_base),
      compat.as_bytes(str(FLAGS.model_version)))
# 第(1)步：构造 SavedModelBuilder 的实例 builder，并设置模型导出路径
print 'Exporting trained model to', export_path
builder = tf.saved_model.builder.SavedModelBuilder(export_path)
# 第(2)步：定义模型服务的 SignatureDef
prediction_signature=...
classification_signature=...
# 第(3)步：使用 builder 导出模型
builder.add_meta_graph_and_variables(
    sess, [tf.saved_model.tag_constants.SERVING],
    signature_def_map={
        'predict_images':
            prediction_signature,
        tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY:
            classification_signature,
    },
    legacy_init_op=legacy_init_op)
builder.save()
```

SavedModelBuilder 构造方法的输入参数 export_path 表示模型导出的目标路径。当 SavedModelBuilder 的实例 builder 构造时，其构造方法内部会创建该目录。如果目录已经存在，那么程序会抛出异常。

创建 SavedModelBuilder 的实例后，我们还需要调用它的 add_meta_graph_and_variables

成员方法，添加期望导出的数据流图和模型参数。add_meta_graph_and_variables 方法的主要输入参数包括 sess、tags 和 signature_def_map。

- sess 是包含待导出模型的 TensorFlow 会话。
- tags 是数据流图的类型标签，可选的取值包括 SERVING、TRAINING 和 GPU，它们分别表示该数据流图用于提供服务、训练模型，以及使用 GPU 设备。
- signature_def_map 以键值对形式保存 SignatureDef 对象，用于记录模型服务提供的不同接口签名。其关键字允许用户自定义（如 predict_images），也允许使用 tf.saved_model.signature_constants 中定义的常量（如 DEFAULT_SERVING_SIGNATURE_DEF_KEY）。signature_def_map 中保存的 SignatureDef 对象会被加入到 SavedModel 中。

我们首先关注图像预测服务。下面的代码展示了图像预测服务的接口签名——prediction_signature：

```
tensor_info_x = tf.saved_model.utils.build_tensor_info(x)
tensor_info_y = tf.saved_model.utils.build_tensor_info(y)

prediction_signature = (
    tf.saved_model.signature_def_utils.build_signature_def(
        inputs={'images': tensor_info_x},
        outputs={'scores': tensor_info_y},
        method_name=tf.saved_model.signature_constants.PREDICT_METHOD_NAME))
```

其中，tf.saved_model.utils.build_tensor_info 方法提供了构建 TensorInfo Protocol Buffers 的方法。该服务的输入和输出均为 TensorInfo 数据类型。方法名称使用 tf.saved_model.signature_constants 中定义的 PREDICT_METHOD_NAME 常量，表示预测服务。

接下来，我们关注图像分类服务。下面的代码展示了图像分类服务的接口签名——classification_signature：

```
classification_inputs = tf.saved_model.utils.build_tensor_info(
    serialized_tf_example)
classification_outputs_classes = tf.saved_model.utils.build_tensor_info(
    prediction_classes)
classification_outputs_scores = tf.saved_model.utils.build_tensor_info(values)

classification_signature = (
    tf.saved_model.signature_def_utils.build_signature_def(
        inputs={
            tf.saved_model.signature_constants.CLASSIFY_INPUTS:
                classification_inputs
        },
        outputs={
            tf.saved_model.signature_constants.CLASSIFY_OUTPUT_CLASSES:
                classification_outputs_classes,
            tf.saved_model.signature_constants.CLASSIFY_OUTPUT_SCORES:
                classification_outputs_scores
        },
        method_name=tf.saved_model.signature_constants.CLASSIFY_METHOD_NAME))
```

其中服务的输入为序列化后的图像样例，输出包括两项——classification_outputs_classes 和 classification_outputs_scores，它们分别表示计算得到的类别和分数（置信度）。方法名称使用 tf.saved_model.signature_constants 中定义的 CLASSIFY_METHOD_NAME 常量，表示分类服务。

最后，执行程序并导出模型。在运行 mnist_saved_model.py 脚本时，builder 实例默认将导出模型的版本号设置为 1。此外，mnist_saved_model.py 文件也定义了模型版本号参数 FLAGS.model_version。用户在执行该脚本时，可以传入 model_version 参数，显式指定导出模型的版本号。为了方便说明，我们约定将模型导出到/tmp/mnist 目录下。下面的命令用于导出训练 1000 步后的 MNIST softmax 模型：

```
$ python tensorflow_serving/example/mnist_saved_model.py --training_iteration=1000
--model_version=1 /tmp/mnist
Training model...
...
training accuracy 0.9092
Done training!
Exporting trained model to /tmp/mnist/1
Done exporting!
```


7.4.2 发布模型服务

成功导出模型后，我们就可以使用 ModelServer 发布模型了。具体方法是执行 tensorflow_model_server 命令，并指定模型名称、模型文件路径和服务端口号等参数。下面的命令演示了如何发布 MNIST softmax 模型服务：

```
$ tensorflow_model_server --port=9000 --model_name=mnist --model_base_path=/tmp/mnist
2017-08-20 22:15:52.623923: I tensorflow_serving/model_servers/server_core.cc:434]
    Adding/updating models.
2017-08-20 22:15:52.623941: I tensorflow_serving/model_servers/server_core.cc:485]
    (Re-)adding model: mnist
2017-08-20 22:15:52.729438: I tensorflow_serving/core/basic_manager.cc:705] Successfully
    reserved resources to load servable {name: mnist version: 1}
2017-08-20 22:15:52.729485: I tensorflow_serving/core/loader_harness.cc:66] Approving load
    for servable version {name: mnist version: 1}
2017-08-20 22:15:52.729502: I tensorflow_serving/core/loader_harness.cc:74] Loading servable
    version {name: mnist version: 1}
...
2017-08-20 22:15:52.757397: I external/org_tensorflow/tensorflow/cc/saved_model/
    loader.cc:284] Loading SavedModel: success. Took 27830 microseconds.
2017-08-20 22:15:52.757596: I tensorflow_serving/core/loader_harness.cc:86]
    Successfully loaded servable version {name: mnist version: 1}
2017-08-20 22:15:52.790754: I tensorflow_serving/model_servers/main.cc:288]
    Running ModelServer at 0.0.0.0:9000 ...
```

从日志结果来看，ModelServer 已将 MNIST softmax 模型服务成功地发布在本地的 9000 端口。为了调用该服务，我们需要编写符合服务接口的 gRPC 客户端代码。针对这个示例性的 MNIST 模型服务，TensorFlow Serving 项目提供了一个 Python 客户端示例，即 tensorflow_serving/example/

mnist_client.py。下面的命令使用该客户端调用已发布的 MNIST softmax 模型服务，请求识别 MNIST 测试数据集中前 1000 张手写体数字图像：

```
$ python tensorflow_serving/example/mnist_client.py --num_tests=1000
--server=localhost:9000
...
Inference error rate: 10.4%
```

因为我们加载的 MNIST softmax 模型在验证集上的准确率为 90% 左右，所以识别结果的平均错误率应该在 10% 左右。从命令输出的结果来看，MNIST softmax 模型服务正确地响应了我们从客户端发起的请求。

综上，我们成功导出了经过训练的 MNIST softmax 模型，并将其发布到了本地的 9000 端口以提供模型服务。我们可以使用多种语言编写的 gRPC 客户端访问该服务，以便对目标数据执行推理（预测）操作。

7.4.3　更新线上模型服务

现在我们已经学会了如何发布特定版本的模型服务。为了能够将更加成熟和精确的模型以迭代发布方式提供给用户，我们有必要了解如何更新已发布的模型服务。这里介绍一种简单的方式，即模型服务自动更新方式。对于模型多次训练之后，新旧版本的文件处于同一目录的情况，ModelServer 默认会自动加载新版本的模型。为了方便说明，我们沿用上一节的例子。在 MNIST softmax 模型的版本 1 已经上线的前提下，我们将模型继续训练到 10000 步后导出版本 2。训练和导出新版模型的命令及其输出如下：

```
python tensorflow_serving/example/mnist_saved_model.py --training_iteration=10000
--model_version=2 /tmp/mnist
Training model...
...
training accuracy 0.9205
Done training!
Exporting trained model to /tmp/mnist/2
Done exporting!
```

可以看出，版本 2 的准确率已提升到 92.05%。

当模型导出完成后，ModelServer 立即感知到/tmp/mnist 目录下生成了新版本的 MNIST softmax 模型文件。此时，ModelServer 会卸载模型的版本 1，并加载版本 2。下面是 ModelServer 更新线上模型服务的日志示例：

```
2017-08-21 22:56:08.412844: I tensorflow_serving/core/basic_manager.cc:705] Successfully
    reserved resources to load servable {name: mnist version: 2}
2017-08-21 22:56:08.412873: I tensorflow_serving/core/loader_harness.cc:66] Approving load
    for servable version {name: mnist version: 2}
2017-08-21 22:56:08.412886: I tensorflow_serving/core/loader_harness.cc:74] Loading servable
    version {name: mnist version: 2}
2017-08-21 22:56:08.412911: I external/org_tensorflow/tensorflow/contrib/session_bundle/
    bundle_shim.cc:360] Attempting to load native SavedModelBundle in bundle-shim from:
```

```
    /tmp/mnist/2
2017-08-21 22:56:08.412932: I external/org_tensorflow/tensorflow/cc/saved_model/
    loader.cc:236] Loading SavedModel from: /tmp/mnist/2
2017-08-21 22:56:08.416698: I external/org_tensorflow/tensorflow/cc/saved_model/
    loader.cc:155] Restoring SavedModel bundle.
2017-08-21 22:56:08.421436: I external/org_tensorflow/tensorflow/cc/saved_model/
    loader.cc:190] Running LegacyInitOp on SavedModel bundle.
2017-08-21 22:56:08.424385: I external/org_tensorflow/tensorflow/cc/saved_model/
    loader.cc:284] Loading SavedModel: success. Took 11454 microseconds.
2017-08-21 22:56:08.424665: I tensorflow_serving/core/loader_harness.cc:86]
    Successfully loaded servable version {name: mnist version: 2}
2017-08-21 22:56:08.517098: I tensorflow_serving/core/loader_harness.cc:137]
    Quiescing servable version {name: mnist version: 1}
2017-08-21 22:56:08.517152: I tensorflow_serving/core/loader_harness.cc:144]
    Done quiescing servable version {name: mnist version: 1}
2017-08-21 22:56:08.517172: I tensorflow_serving/core/loader_harness.cc:119]
    Unloading servable version {name: mnist version: 1}
2017-08-21 22:56:08.518784: I ./tensorflow_serving/core/simple_loader.h:294]
    Calling MallocExtension_ReleaseToSystem() after servable unload with 60651
2017-08-21 22:56:08.518809: I tensorflow_serving/core/loader_harness.cc:127]
    Done unloading servable version {name: mnist version: 1}
```

模型服务自动更新的实现方法比较简单，但其灵活性较差。程序一旦导出模型，ModelServer便会立刻更新服务，这不一定完全符合实际使用场景。在模型训练过程中，我们可能希望不断地尝试不同的训练参数，只有在得到相对满意的模型之后才将其发布。这种做法不但能够保证模型本身的质量与精度，同时也有助于确保模型服务的稳定性与可靠性。针对这种场景，我们需要修改服务的版本更新策略，显式控制 ModelServer 上的服务更新。具体方法请读者参考 TensorFlow Serving API 文档。

7.5 小结

TensorFlow Serving 作为一套模型托管工具，打通了从模型训练到服务发布的工作流水线，补全了在生产环境中部署深度学习和机器学习模型的后期环节。针对推理（预测）态服务的特点，TensorFlow Serving 提出了一套持续训练、持续集成和持续发布的标准化流程，并提供了相应的插件和工具。借助 gRPC 和 Protocol Buffers 的能力，TensorFlow Serving 发布的服务天然具有跨平台、支持多语言客户端的特点。同时，TensorFlow Serving 还兼具易用性、高效性和灵活性优势。熟练掌握并灵活应用 TensorFlow Serving，是将算法模型推向实用系统的一条便捷之路。

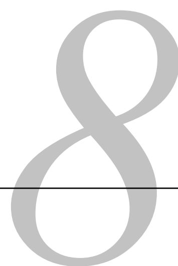
第三部分

算法模型篇

- ❑ 第 8 章　深度学习概述
- ❑ 第 9 章　CNN 模型
- ❑ 第 10 章　GAN 模型
- ❑ 第 11 章　RNN 模型

第 8 章

深度学习概述



学会了 TensorFlow 的基本原理和使用方法之后，我们即将进入实战环节。TensorFlow 的主要应用场景是训练和部署深度学习模型，从而解决感知、推理、决策等人工智能及相关领域的具体问题。了解深度学习的历史与现状、认识一些经典的神经网络模型，是独立开发应用专属模型的前提。本章对深度学习技术的来龙去脉进行介绍，然后展示一组典型的应用场景，借此背景知识，将读者带入人工智能算法开发的广阔天地，引导读者充分施展 TensorFlow 作为工具的潜能。

8.1 深度学习的历史

大多数数据科学家和算法工程师对神经网络并不陌生。深度学习是神经网络的进一步延伸。通过加深模型深度，深度学习可以从数据中学习到不同程度的抽象特征。在深度学习出现之前，当人们要处理和分析非结构化数据（如语音、文本和图像等）的时候，需要先利用人工设计的特征提取模块从原始数据中抽取特征，然后再将特征输入到分类器或者回归器（如 SVM 等）以得到最终的结果。这样做的弊端在于人工设计的特征很难应对各式各样的输入数据和应用场景。深度学习可以自动地从原始数据中学习特征，避免了人工特征抽取。现在深度学习已经在图像识别、语音识别、机器翻译等任务中已经取得了一系列突破。本节主要从最早的感知机模型与神经网络开始，介绍深度学习的发展历史。

8.1.1 感知机模型与神经网络

受 Warren McCulloch 和 Walter Pitts 在神经元建模方面工作的启发，心理学家 Frank Rosenblatt 于 1957 年发明了神经感知机模型 Perceptron，然后将其用于解决分类问题。如图 8-1 所示，该模型的设计参考了大脑中神经元传递信号的工作机制。在这个模型中，输入神经元的信号由向量 $\boldsymbol{x}$ 表示（向量中每个值为 0 或 1），模型参数（也称突触强度）由向量 $\boldsymbol{w}^{\mathrm{T}}$ 表示。向量 $\boldsymbol{x}$ 和 $\boldsymbol{w}^{\mathrm{T}}$ 做线性乘积后，再经过一个非线性激活函数，就得到了输出神经元的信号值 y。其中，非线性激活函数采用的是简单的阈值函数。在这种阈值函数中，当输入大于阈值 T 时，输出为 1，否则为 0。因此，这是一个二分类模型。

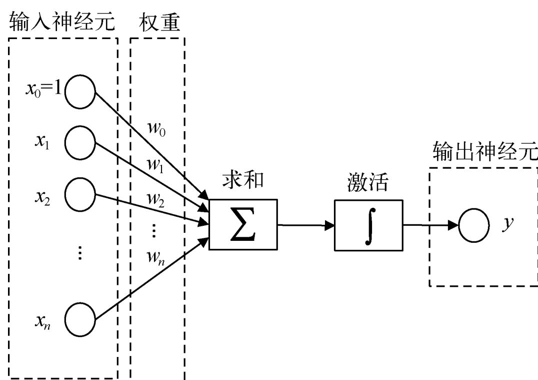


图 8-1　神经感知机模型的示意图

如何训练模型参数是一个重要问题，最原始的训练方法如下所示。

(1) 随机初始化模型参数 $\boldsymbol{w}^{\mathrm{T}}$。

(2) 计算当前训练样本所对应的输出神经元的信号值 y。将输入向量 $\boldsymbol{x}$ 乘以 $(\bar{y}-y)\eta$ 作为 $\boldsymbol{w}$ 的更新量，其中 $\bar{y}$ 为训练样本对应的真实标签值，η 为一个大于 0 的学习速率。

(3) 重新训练下一个样本，并根据步骤 (2) 来更新模型，直至模型收敛。

这个训练方法本质上就是监督学习中常用的梯度下降法。Frank Rosenblatt 将这个神经感知机模型在康奈尔航空实验室的 IBM 704 计算机上实现了。但这个感知机模型只可以解决二分类问题。如果要解决多分类问题，需要多个输出神经元组成输出层进行分类。这种形式的模型就是最简单的神经网络模型。通过堆叠更多层的神经元，并将相邻层的神经元相互连接，可以得到更复杂的神经网络。Frank Rosenblatt、Warren McCulloch 和 Walter Pitts 等人都是这种连接主义的支持者，他们坚信这种基于感知机的模型可以解决人工智能所面临的复杂问题，比如语音识别、图像识别等。

8.1.2　神经网络的寒冬与复苏

1956 年，在 MIT 教授 Marvin Minsky 等人发起的达特茅斯会议上，人工智能这个概念被首次提出。当时人工智能的研究者倾向于用严格的数学逻辑推理生成规则，并基于这些规则操作数学符号生成人工智能算法。Marvin Minsky 比较怀疑感知机模型能否实现真正意义上的人工智能。他证明了感知机只可以实现“与”“或”的逻辑操作，而实现不了“异或”操作，也就是说感知机模型难以解决非线性可分问题，并且当时业界几乎没有训练多层感知机和神经网络的可行方法。正是由于以 Marvin Minsky 为代表的很多人工智能界学者的强力反对，感知机和神经网络的研究在开始不久后受挫，当时的研究机构很难申请到相关的项目经费。神经网络的发展开始进入寒冬。

尽管如此，还是有一些学者在感知机的基础上，继续坚持研究多层神经网络模型及其训练算法。现在，大多数算法开发者都已经知道多层神经网络可以用 BP（Back Propagation，后向传播）算法来训练模型参数。该算法的基本思想是通过损失函数对模型参数求导，并根据复合函数求导

常用的“链式法则”将不同层的模型参数的梯度联系起来，使得计算所有模型参数的梯度更简单。BP 算法的思想早在 20 世纪六七十年代就被提出来了。David Parker、Yann LeCun 等深度学习研究者都相继提出利用 BP 算法来训练多层神经网络，然而当时并没有引起整个学术界足够的重视。直到 1986 年，David Rumelhart 和 Geoffrey Hinton 等人发表了一篇后来成为经典的论文①，它清晰地描述了 BP 算法的框架，这才使得 BP 算法真正流行起来，并带来了神经网络在 80 年代的辉煌。多层神经网络被证明可以模拟任意的非线性函数。人们可以通过 BP 算法训练多层神经网络模型中的所有未知参数，以实现特征的自动抽取。

神经网络最开始的应用是识别手写数字。日本东京大学教授福岛邦彦（Kunihiko Fukushima）发明了多层神经网络 Neocognitron 用于手写数字和字母识别②。受 Neocognitron 的启发，当时还是贝尔实验室研究员的 Yann LeCun 等人发明了多层 CNN（Convolutional Neural Network，卷积神经网络）模型，该网络的基本结构如图 8-2 所示。在网络最初的几层（即卷积层）中，就单通道的输入和输出而言，每层的输出神经元只与输入神经元的部分相连接，并且这些连接所对应的权重可以被所有的输出神经元共享。

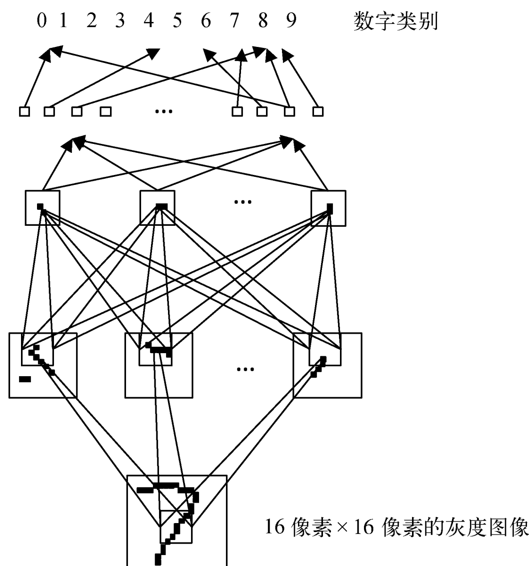


图 8-2　用于手写数字识别的 CNN 示意图

CNN 模型常用于图像等数据的特征自动提取。图像中很多初级特征都满足局域性和位置无关性，这给 CNN 的设计带来了灵感。局域性是指图像中某个区域的特征只与以该区域为中心的

① D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. Nature, 1986, 323: 533–536

② K.Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. Neural Networks, 1988, 1(2): 119-130

局部范围内的像素值有关，这使得两层神经元之间的稠密连接可以变为局域稠密连接（从全局的角度看，稠密连接变为了稀疏连接）。位置无关性是指输出神经元提取的特征对于像素位置是无关的。比如，假设某一层的卷积计算是用来提取图像中边缘信息的，那么提取边缘的操作（由连接的权重定义）应该是在所有像素位置共享的一个操作。因此，卷积带来的好处是前后两层神经元之间的连接数大大减少了，而且模型参数可以共享。这不仅使得模型训练成为可能，而且能防止过拟合。

20 世纪 90 年代期间，Yann LeCun 等人基于 CNN 实现了手写数字识别系统，应用于美国部分银行支票上数字的识别，实现了商用化。BP 算法使得多层神经网络训练变得简单，也促使了手写数字识别等系统的商业成功。至此，神经网络的发展经历了第一次寒冬和复苏。

8.1.3 神经网络的发展与第二次寒冬

BP 算法是训练大多数神经网络的通用算法。除了 CNN，很多其他形式的神经网络模型也可以被很好地训练出来，其中具有代表性的就是自编码器。自编码器属于无监督学习中的一种，可以训练无标注的数据。

如图 8-3 所示，自编码器的基本思想是用隐藏层中少量的神经元来表示输入数据，以学习到更加抽象的特征。为了以无监督的方式训练编码器，我们需要再增加一个解码器，实现隐藏层到输出层（输出层与输入层的维度相同）的映射，并将输入层和输出层之间的误差作为损失函数进行网络训练。

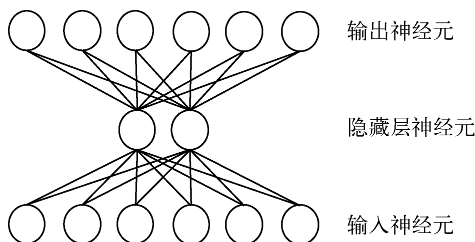


图 8-3 用于特征学习和压缩的自编码器模型示意图

1985 年，Geoffery Hinton 等人提出了玻尔兹曼机及其训练算法。不同于之前的神经网络模型，玻尔兹曼机是一种基于能量函数的模型，源自统计物理学。它以最小化能量函数为目标，模型参数的学习过程带有一定的概率性。如图 8-4 所示，相比传统的单层或多层神经网络，玻尔兹曼机没有明显的分层，其模型内部的神经元是彼此互相连接的。玻尔兹曼机也可以被认为是一种图模型，其中每一个神经元的输出（取 0 或者取 1 的概率值）都取决于和它相连的神经元的激活值及其相应的权重。玻尔兹曼机中所有神经元的输出的概率值形成了一个概率分布。

在玻尔兹曼机中，任何神经元都可以作为可见神经元（visible neuron）或者隐藏神经元（hidden neuron）。在可见神经元和隐藏神经元中，任意一方的值确定后，对方的概率分布即可被计算出

来。因此，玻尔兹曼机不仅可以用来生成隐藏层的概率分布，还可以用来生成输入数据（由可见神经元表示）的概率分布。这意味着玻尔兹曼机既可以是判别模型，也可以是生成模型。针对玻尔兹曼机不容易训练的问题，Geoffery Hinton 等人对其进行了简化，提出了 RBM（Restricted Boltzman Machine，受限玻尔兹曼机）。在 RBM 中，隐藏层的神经元之间没有相互连接，输入神经元之间也没有相互连接，只有隐藏层和输入层的神经元之间存在连接。多层 RBM 或多层自编码器可以堆叠并形成 DBN（Deep Belief Network，深度信念网络）。除了以玻尔兹曼机为代表的随机神经网络之外，大多数神经网络模型仍采用类似于 BP 的算法做训练。例如，在 CNN 用于图像识别的同时，人们也在积极探索用于语音识别的神经网络模型。语音数据与图像数据主要的不同点在于它具有时间序列特性。RNN（Recurrent Neural Network，循环神经网络）是一种以时间序列数据（比如由 10 个单词组成的一个句子）为输入的模型，其典型结构如图 8-5 所示。

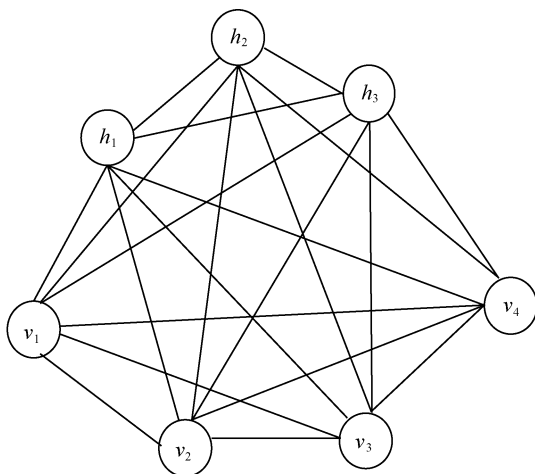


图 8-4 玻尔兹曼机示意图（v 表示可见神经元，h 表示隐藏神经元）

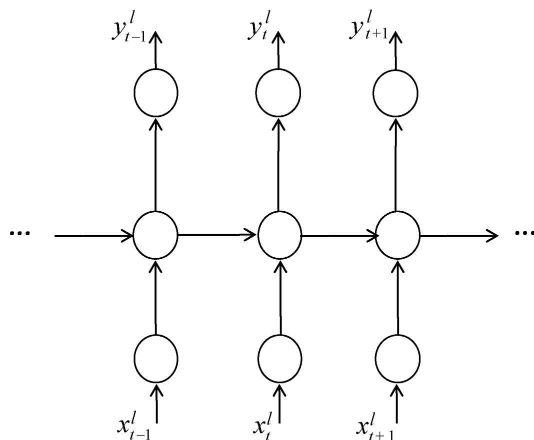


图 8-5 RNN 模型示意图

在 RNN 模型中，隐藏层神经元状态不仅可以输入到下一层，而且可以输入到下一时刻的隐藏层。不同于 CNN 模型，RNN 模型需要采用 BPTT（Back Propagation Through Time，随时间反向传播）算法来训练，其参数梯度在深度方向和时序方向上同时被传播。后来，针对 RNN 难以学习到输入数据中长时间依赖关系的问题，Jürgen Schmidhuber 等人发明了 LSTM（Long-Short Time Memory，长短期记忆）模型。LSTM 模型的整体架构类似于 RNN 模型。它引入“细胞状态”（也称“记忆状态”）来表示时序上的中间信息流，并通过设计 4 个“门”（每个“门”可以看作某种矩阵运算）来控制细胞状态的值。细胞状态有助于改善 RNN 模型常遇到的长时间依赖问题。

随着模型种类的丰富多样化，神经网络所覆盖的应用场景也越来越多。然而，如果我们把模型变得稍微复杂一些，即使有 BP 或 BPTT 算法，也很难训练出一组最优的模型参数。梯度弥散（或梯度消失）是 BP 或 BPTT 算法用于复杂模型时常见的问题。梯度在 BP 或 BPTT 传播过程中需要不断地乘以每层的权重系数和激活函数的导数等。当梯度所乘项小于 1 时，梯度会越变越小，出现梯度弥散问题。对于 CNN 模型，当深度增加时，训练难度会增加；对于 RNN 模型，当输入时间序列的长度或模型深度增加后，训练难度也会增加。因此在 20 世纪 90 年代中期，人们发现没有好的算法能够保证训练出一个好的深度模型。此外，当时的数据量并不丰富，并且计算机的计算能力较弱，难以支撑具有一定复杂度的模型的训练，这使得很多复杂的神经网络模型都难以被真正应用起来。而与此同时，SVM（Support Vector Machine，支持向量机）的兴起吸引了大家的注意力。故而到 20 世纪 90 年代中后期，神经网络模型再度走向寒冬，以 SVM 为代表的机器学习方法由于其实用性高而成为当时流行的方法。

8.1.4 深度学习时代的到来

从 20 世纪 90 年代开始到 2000 年初的这段时间，坚持研究神经网络的学者非常少。当时美国政府或企业已经很少会拿出经费支持神经网络的研究，甚至主流的学术刊物都不太愿意接受神经网络相关的论文。能够坚持下来的只有 Geoffrey Hinton、Yann LeCun、Yoshua Bengio 等少数学者。幸运的是，以 CIFAR（Canadian Institute for Advanced Research）为代表的研究机构支持研究者做一些长远的基础研究，其中就包括了神经网络的研究。2006 年，Geoffrey Hinton 等人发明了一种深度网络训练方法。这种方法首先采用无监督学习对每层参数做初始化，再用监督学习做参数训练。这实质上是一种半监督学习方法。Geoffrey Hinton 等人将这种逐层训练的方法用于图像识别，取得了很好的效果。同时，他们也意识到用多个 GPU 设备训练深度模型可以提升训练速度。当时工业界的一些巨头公司（如 Microsoft、Google 等）已经意识到深度学习模型在语音识别、图像识别场景中有巨大的潜力，故而纷纷开始重点研究深度学习模型。

2011 年，斯坦福大学的副教授吴恩达（Andrew Ng）找到了 Google 公司的 Jeff Dean 等人，他们共同讨论了如何利用 Google 庞大的计算资源来快速训练深度神经网络，然后发起了 Google Brain 项目。他们当时搭建了一个分布式的异步深度神经网络模型训练平台——DistBelief，其中采用了 16000 多个 CPU 训练一个具有 10 亿参数量的深度神经网络。这个神经网络模型通过对大

量 YouTube 视频数据的学习，最终能够自动地识别视频中的猫。

2012 年，Geoffrey Hinton 和他的两个学生在机器学习与人工智能领域的顶级会议 NIPS（Neural Information Processing Systems）上发表了震惊计算机视觉和人工智能学界的 AlexNet 模型，从此正式拉开了深度学习时代的序幕。AlexNet 模型主要由 5 个卷积层、3 个全连接层和其他一些层组成。在开源的图像数据集 ImageNet 上，AlexNet 模型以 84.7% 的 Top-5 精度获得图像分类竞赛冠军，领先第二名约 10 个百分点。这在会议上引起了不小的轰动，也标志着深度学习在图像分类方面实现了里程碑式的跨越。事实上，在 AlexNet 模型出现之前，每年很多计算机视觉专家和学者都会提出各种基于人工定义的特征提取算子和图像分类模型，但是最终精度的提升并不是很明显。当时人们认为在 ImageNet 数据集上能够提升 1 个百分点就已经是非常了不起的进步了。虽然在 2012 年之前，吴恩达、余凯（地平线机器人公司创始人）等人一直在坚持使用神经网络对图像特征进行自动学习，但是由于缺乏足够的理论支撑和实际实验效果的证明，他们没有被当时计算机视觉界的主流学者认可。直到 AlexNet 模型出现后，计算机视觉研究者对神经网络和深度学习的看法才得到颠覆性的改变。

从 2012 年开始到现在的短短几年时间内，无论在学术界还是工业界，深度学习都出现了爆炸式的发展。人工智能要解决的问题主要包括感知、推理、决策和联想等。当前深度学习模型在处理感知问题上相对比较成熟。比如，在大规模的测试集下，深度学习识别图像或语音的准确率可以做到很高，甚至在有些场景下超过人类的识别准确率。深度学习在自然语言处理上的应用也取得了不少进步。Microsoft、Google、Facebook、百度等公司已经宣布深度学习模型在机器翻译上的表现要明显好于其他模型。深度学习和强化学习的结合（即深度强化学习）有望攻克推理、决策等更高级的人工智能问题。已被 Google 收购的 DeepMind 公司推出的 AlphaGo 机器人正是采用了深度强化学习技术来学习如何下围棋，并在 2016 年 3 月与韩国顶级围棋选手李世乭的比赛中以总比分 4∶1 取得胜利。2017 年 5 月，第二代 AlphaGo 机器人与围棋世界第一成绩的保持者柯洁比赛，以总比分 3∶0 取得了胜利。

在深度学习模型中，模型的复杂度会随着网络深度的增加而成指数增加。随着大数据时代的到来，以及深度学习模型设计和训练方法的改进，我们可以训练更深、更复杂的模型，同时避免过拟合现象。另外，GPU 等硬件设备和分布式集群计算能力的增强，使得深度学习模型的训练时间大幅降低。这些因素共同促进了当前深度学习的广泛应用。

相信在接下来的几年中，深度学习和其他学习策略（如对偶学习、迁移学习等）的结合，以及新型深度学习网络模型（如记忆网络、复数深度学习等）的发明会带来更多的突破性技术，进而对人工智能在各个行业的应用产生广泛而深远的影响。

8.2　深度学习的主要应用

深度学习模型特别擅长拟合复杂的非线性函数，这使得它可以用于很多非结构化数据的特征提取（或特征学习）。这些非结构化数据包括计算机视觉中的图像视频、自然语言处理中的句子

和强化学习中的状态（可能是图像或者其他）等。下面分别介绍深度学习在计算机视觉、自然语言处理和强化学习中的应用。

8.2.1 计算机视觉

计算机视觉是一门交叉学科。它融合了物理学、信息学、计算机科学、人工智能等多项技术，并广泛应用于多种领域，如工业制造、医学辅助诊断、视频监控等。计算机视觉研究的重点之一是从图像和视频中获取人们感兴趣的信息或知识。常见的计算机视觉任务包括图像分类、图像匹配、物体检测、目标跟踪、视频分析与理解等。这些任务共同的难点在于图像特征提取。特征提取的传统做法是人工设计各种图像特征提取算子，如 SIFT（Scale-Invariant Feature Transform）、LBP（Local Binary Pattern）、HOG（Histogram of Oriented Gradients）等，并将这些算子应用于某个具体的计算机视觉任务。为了适应不同的任务需求，算法开发者需要人工设计不同的特征提取算子和相应的图像处理模型。然而，人工设计算子或模型精度往往不会很高，而且通常会有一些限制，尤其对光照、遮挡等现象的健壮性较差。

在 20 世纪 90 年代，CNN 模型已经开始用于数字和字母识别。2012 年，Geoffery Hinton 领导的团队利用深度卷积神经网络取得了大规模图像分类领域内里程碑式的突破。到现在为止，CNN 模型的深度还在不断增加（例如，有些模型已经超过 1000 层）。表 8-1 列出了近年来常用的 CNN 模型的精度随深度的变化趋势（其中，图像分类精度参考 ILSVRC 竞赛结果，物体体测精度参考相关论文和部分开源实现）。可以看出，在一般情况下模型精度会随深度增加而增加。在 ImageNet 图像分类方面，深度学习的识别精度已经超过了人眼。在图像检测、图像分割等方面，深度学习的精度也在不断提高。

表 8-1 近年来常用的 CNN 模型的精度随深度变化的趋势

任务类型	深度卷积神经网络模型	深度（层数）	精　　度
图像分类	VGG	19	92.7% (Top5 accuracy)
图像分类	GoogLeNet	22	93.3% (Top5 accuracy)
图像分类	ResNet	152	96.43% (Top5 accuracy)
物体检测	RCNN + AlexNet	8	58% (mAP)
物体检测	Faster RCNN + VGG16	16	71% (mAP)
物体检测	Faster RCNN + ResNet101	101	76% (mAP)

深度 CNN 模型使得端到端的图像处理和分析成为可能。比如，我们可以构造一个输入为图像、输出为感兴趣目标（包含其位置、大小等量化信息）的深度卷积网络模型，并设计特定的损失函数和训练策略。该模型可以将原始图像像素直接映射到最终的物体检测结果。近年来流行的 SSD（Single Shot MultiBox Detector）物体检测模型就是这样做的。深度 CNN 模型灵活多变，我们可以通过组合不同的子网络、构造相应的损失函数和设计特定的训练方法，得到更复杂的模型。例如，Faster RCNN 模型就由多个网络构成，它包括基础网络（用于特征提取）、RPN（Region Proposal Network，用于提取物体候选框）、分类网络和回归网络（用于对候选框位置和大小做回

归，以及对候选框中的物体做分类）。

另外，深度卷积神经网络提取的特征具有很好的通用性。如果我们要在一份新的数据集上训练模型，那么可以将已在其他数据集（如 ImageNet）上训练好的模型参数作为新模型参数的初始值，然后只训练该模型的最后几层即可。这说明深度卷积神经网络具有迁移学习的特点，即一个数据集上训练好的模型可以通过微调迁移用在另外一个数据集上。特别是当新的数据集中训练数据较少时，我们通过这种微调的方式可以很容易得到一个精度较高的模型。

目前，深度卷积神经网络与各类计算机视觉任务和应用场景的不断融合，逐渐催生出各类卷积神经网络模型框架。虽然深度卷积神经网络在计算机视觉中的应用越来越广泛，但是其计算复杂度高。如何提升计算速度并降低计算功耗就显得尤为重要，这也是当前深度学习领域很重要的研究方向。

8.2.2　自然语言处理

自然语言处理是一门综合了语言学、数学与计算机科学的交叉学科，其目的是让计算机能够理解人类的语言，并根据语言中的语义信息去执行其他工作。自然语言处理的任务从易到难主要包括：拼写检查、关键词提取、词语查找和替换、机器翻译、智能问答等。自然语言处理的难点在于语言的多义性和语法的复杂性。

传统的自然语言处理方法需要手工提取特征、规则并进行句法分析。比如，WordNet、ConceptNet、FrameNet 这些经典的模型主要通过手工方法建立起词与词之间的语义关系。因为词语或短语的种类非常多，其组合更多，所以计算机很难遍历所有的情况，这也正是自然语言处理的难点之一。随着大数据时代的到来，深度学习作为自动表征学习的代表性技术，可以自动提取单词、短语、句子中的特征并将其量化表示，甚至可以端到端地完成一个自然语言处理任务。例如，Seq2Seq 模型可以用于机器翻译，并且效果比传统模型好。

自然语言处理的一项基础工作是将句子中的每个单词向量化表示以方便后续处理。最简单的一种方式是用 One-Hot 编码形式表示每个单词。假设词汇表有 1000 个词，那么每个词需要用 1000 维向量表示。根据单词在词汇表中的索引位置确定该 1000 维向量中哪个值为 1。然而，这种表示方法中单词之间是两两正交的，我们无法得到词语之间的相似度，也就无法判别其语义信息。后来，有人提出采用 word-documents 方式建立矩阵，用每个词语在文档中出现的次数来表示词向量。还有人提出基于 co-occurrence matrix 的方法：先建立一个词语矩阵（对称矩阵），行列都依次为每个词语；然后遍历整个语料库，对相邻两个同时出现的词在词语矩阵相应的位置上加 1；最后基于该词汇矩阵做 SVD（Singular Value Decomposition）以得到词向量。不过，这些方法都有比较大的问题：首先，这些矩阵都是很高维度而且稀疏的，训练比较困难；其次，如果词汇表增加，就需要更新矩阵并重新训练词向量模型，代价很高。

神经网络模型用于词向量表达（将一个词用 K 维实数的向量表示，且 $K \ll |V|$，其中 $|V|$ 为词汇表大小）可以避免传统词向量模型的缺点。CBOW（Continuous Bag-of-Words）和 Skip-Gram

是常用于词向量表示的经典模型，其基本思想和自编码器比较类似。在 CBOW 模型中，首先假设存在一个未知的词向量生成矩阵 $\boldsymbol{W}$，且每个词语（其词向量为 $\boldsymbol{V}(x)$）前后的几个词（这些词的词向量为 $\boldsymbol{V}(1),\cdots,\boldsymbol{V}(x-1),\boldsymbol{V}(x+1),\cdots,\boldsymbol{V}(n)$）组成了该词的上下文。然后根据此上下文中每个词经 $\boldsymbol{W}$ 变换后的词向量得到一个平均向量，再根据这个平均向量预测该词，通过迭代训练来学习 $\boldsymbol{W}$ 中的未知参数。这种方法训练出来的词向量模型具有较强的上下文语义关系。而 Skip-Gram 与 CBOW 正好相反，它是利用当前词来预测上下文的。

以句子中每个单词的词向量为基础，我们可以计算出每个单词在整个语料库中出现的概率，然后通过计算所有单词的联合概率得到整个句子的概率，从而建立统计语言模型。假设一个句子中的每个单词对应的词向量依次为 $\boldsymbol{V}(1),\cdots,\boldsymbol{V}(n)$，那么整个句子的概率为：

$$P=P(\boldsymbol{V}(1),\cdots,\boldsymbol{V}(n))=P(\boldsymbol{V}(1))P(\boldsymbol{V}(2)|\boldsymbol{V}(1))\cdots P(\boldsymbol{V}(n)|\boldsymbol{V}(n-1),\boldsymbol{V}(n-2),\cdots,\boldsymbol{V}(1))$$

一个句子的概率越高，这个句子就越有意义。然而，正如上述公式所示，计算一个句子中所有单词的联合概率是非常复杂的。为了简化计算，通常我们都采用 n-gram 语言模型。在 n-gram 模型中，计算每个词的条件概率时，我们需要用到该词的前 n 个词。当 n=1 时，整个句子的概率简化为：

$$P=P(\boldsymbol{V}(1),\cdots,\boldsymbol{V}(n))=P(\boldsymbol{V}(1))P(\boldsymbol{V}(2)|\boldsymbol{V}(1))\cdots P(\boldsymbol{V}(n)|\boldsymbol{V}(n-1))$$

Yoshua Bengio 等人于 2003 年提出了一种使用神经网络同时训练词向量和语言模型的方法，这是将神经网络用于自然语言处理的经典工作之一。该方法的实现机制如图 8-6 所示。在计算每个词语的上下文关系时，该模型可以捕捉到更广范围内的词语间依赖关系。

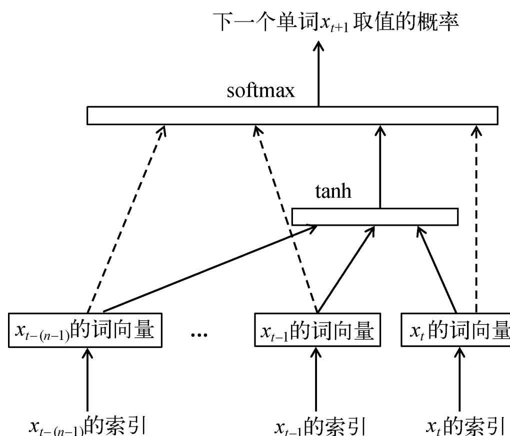


图 8-6 Yoshua Bengio 等人提出的词向量学习和语言学模型

然而，上述语言模型本质上仍是 n-gram 模型，只能根据前几个词预测到下一个词（一般 n-gram 模型中 n 比较小）。现在大部分语言模型都基于 RNN 模型实现，如图 8-7 所示。RNN 模型中隐藏层状态的横向传递使得更早之前的单词信息被利用起来。

如 8.1.3 节所述，如果作为 RNN 输入的时间序列（句子等）的依赖关系较长，那么用 BPTT

算法训练模型时就会出现梯度弥散现象。LSTM 可以在一定程度上改善梯度弥散现象。修改 LSTM 基本单元内部的控制门和控制边，或者融合细胞状态和隐藏状态，可以构造出很多 LSTM 的变种，比如在神经机器翻译上常用的 GRU（Gated Recurrent Unit）。

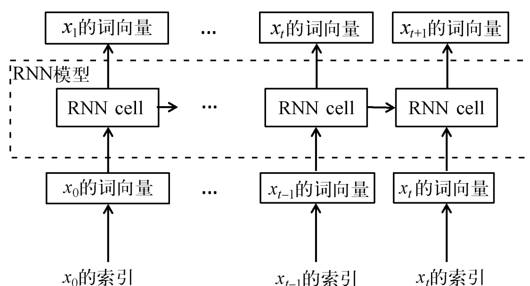


图 8-7　基于 RNN 的语言学模型

不同于以条件随机场（CRF）和隐马尔可夫模型（HMM）等为代表的传统自然语言处理模型，神经网络和深度学习可以同时完成词向量和语言模型的学习，从而实现端到端的学习。例如，目前以 RNN、LSTM 等为基础的深度学习模型已经成功应用于 Google、Microsoft、百度等企业的机器翻译系统，它们的精度远超传统统计翻译模型。随着 RNN、LSTM 模型层数的增加，模型的特征学习能力也会增加。当训练数据较多的时候，我们可以采用多层 RNN、LSTM 模型以提升训练精度。但一般而言，RNN、LSTM 模型的层数不会像 CNN 那样有几十层或几百层。当前业界最新的一些用于机器翻译的 RNN、LSTM 模型的层数大多在 10 层以内。

此外，以多层 RNN、LSTM 等为代表的深度学习模型还用于自然语言对话、问答系统、视频摘要等任务。需要注意的是，CRF 等经典模型有很好的概率图模型理论做支撑，所以人们也常将 RNN 或 LSTM 模型与 CRF 等结合起来使用。

8.2.3　强化学习

机器学习算法通常分为三大类：监督学习、非监督学习和强化学习。不同于其他两类算法，强化学习是一个动态的过程。在强化学习算法中，存在一个 Agent（智能体）及其所在的 Environment（环境）。在时刻 t，Agent 从 Environment 中获得观测信号 $O(t)$，并计算出下一步的执行动作 $A(t)$。同时，Agent 得到 Environment 的回报信号 $R(t)$。Environment 在被 Agent 的动作 $A(t)$ 影响后，再发出下一步的观测信号 $O(t+1)$ 和回报信号 $R(t+1)$。如图 8-8 所示，在 Agent 与 Environment 相互作用的过程中，Agent 的模型得到不断训练。强化学习可以用于机器人自动控制、金融投资、游戏博弈、推荐系统、多轮对话（智能助理）等场景。

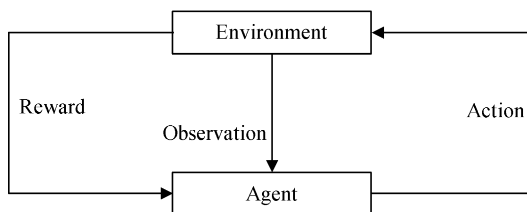


图 8-8　强化学习中 Agent 和 Environment 的交互过程

强化学习算法主要包含三部分模型。

- **Agent 策略模型**：其输入是观测信号，输出是动作。
- **Agent 值函数模型**：用于计算未来奖惩信号累计值的期望，并评价 Agent 的状态和行为的好坏。其中，状态是指该时刻前的一系列观测信号、回报信号和动作所组成的序列的函数，而值函数通常是指当前状态下对未来回报的期望值。
- **Environment 模型**：用来预测环境在一定状态下接收到一定的动作信号后，所产生的新状态和回报信号。

Agent 通过迭代方式学习到策略模型和值函数模型。在策略模型中，Agent 根据当前观测信号选择最优的动作，使得后续得到的回报信号期望值最大。在值函数模型中，Agent 根据当前的观测信号和采取的动作预测将会得到的回报信号。在复杂的强化学习算法中，策略模型和值函数模型都是高度非线性的复杂模型。

深度学习通过多层网络可以模拟非常复杂的非线性函数，且能够处理很高维的输入（如图像、视频），因此可以用在策略模型和值函数模型中。例如，在 Q-Learning（一种常用的强化学习算法）中，基于深度学习的值函数模型为 Deep Q-Network。Deep Q-Network 的输入为当前状态和执行的动作，输出为所期望得到的最终回报。在相应的策略模型中，最优策略是使得 Deep Q-network 输出最大的动作策略。可以将 Q-Network 的求解描述成一个迭代式：$Q(s, a) = \alpha \cdot (r + \gamma \max_{a'} Q(s', a')) + (1 + \alpha) \cdot Q(s, a)$，其中 a、s、r 分别为动作、状态和回报信号，γ 为折扣因子，α 为学习速率。当前 Q-Network 的输出取决于当前回报值和在最优动作下 Q-Network 的未来输出值。

很多强化学习算法，如 Actor-Critic、DDPG（Deep Deterministic Policy Gradient）都使用深度学习实现策略模型和值函数模型，以提升学习的稳定性并挑战困难的任务。当前强化学习研究的热潮正是传统强化学习和深度学习的结合所带来的。深度学习的出现使得强化学习可以实现真正的端到端训练。比如，在自动驾驶中，输入是摄像头拍摄的实时画面（即当前状态），输出即为方向盘的控制方向或者车速大小。传统的模型难以处理图像、视频这一类非常高维的非结构化数据，因此人们常借助深度 CNN 模型来提取特征。以 OpenAI、DeepMind 为代表的机构或公司正在加大对深度强化学习的投入，它们开放了各自的深度学习算法测试环境（如 OpenAI Gym）。随着大数据、深度学习、分布式计算、开源训练平台和测试环境的发展，深度强化学习将给人工智能中一些高级问题的解决带来更强的驱动力。

8.3 深度学习与 TensorFlow

近年来，深度学习模型在语音和图像识别准确率上的突破彻底引爆了这一领域。短短几年的时间内，深度学习热潮席卷了工业界和学术界。它在图像、视频、语音、语义、对话和翻译等方面取得的发展，主要得益于大数据、神经网络、优化算法、高性能计算设备（GPU 等）、分布式计算等技术的进步。除此之外，开源软件生态系统也极大地促进了深度学习领域的发展。这些软件使得新想法得以快速实验，同时使得更多的数据科学家和算法工程师参与到深度学习的研究和应用开发中，形成了良性循环。另外，在当前深度学习的可解释性理论研究比较滞后的情况下，研究人员更加需要高效的深度学习开发工具以快速获得实验结果，加速理论的验证。从深度学习领域的特点出发，它对计算引擎的需求主要体现在以下几个方面。在这几个方面，TensorFlow 均提供了强有力的支撑。

- 深度学习所涉及的模型、算法在底层大多可以转换为多维矩阵或张量计算。因此，深度学习引擎首先必须是一个优秀的张量计算库。一般情况下，随着数据量的增加，深度学习模型的复杂度可以变得更高，从而使得模型的准确率进一步提升。这就意味着深度学习模型对于矩阵计算速度的要求越来越高。作为一款异构计算引擎，开源的 TensorFlow 版本已经支持 CPU、GPU 两种计算设备。在深度学习模型训练过程中，CPU 通常用于数据读取、预处理、网络通信等，GPU 主要用于密集的张量计算。用户还可以利用 TensorFlow XLA 将深度学习模型部署在其他硬件设备（如 AI 专用芯片等）上，以进一步加速计算。
- 深度学习模型仍在快速发展，而且 CNN、RNN 等模型也在不断彼此融合。即使对于独立的 CNN 或 RNN 模型，模型内部也有很多算子层面的变化，比如 CNN 模型的每一层内可以有多个分支，RNN 模型的每个单元也包括多个复杂的矩阵运算。因此，深度学习引擎必须要足够灵活以支撑更多的模型及其变种。TensorFlow 引擎的算子很丰富，并且用户可以方便地添加自定义算子，这使得 TensorFlow 可以自由地表达任意的深度学习或其他机器学习模型。
- 深度学习模型的训练主要依赖于经典的后向传播算法。为了训练模型，每一层参数的梯度都要被计算，但是随着模型层数的增加，直接计算参数的梯度过于复杂。在后向传播算法中，误差的逐层传递可以使得每层参数的梯度计算变得非常简单。这种逐层传递的计算所依赖的正是数学中函数求导时常用的链式法则。TensorFlow 引擎支持自动微分，用户只需要写好数据流图的前向计算过程，TensorFlow 运行时就能自动生成数据流图的后向计算部分。这使得用户不需要关心复杂的后向计算过程，从而能够专注于设计模型的架构。
- 深度学习模型的训练是一个极耗资源的过程。虽然 GPU 或 AI 专用芯片的计算能力在逐年提升，但是单个计算设备的能力仍然有限。因此，分布式训练对于深度学习而言非常有必要。训练效率的提升有助于模型或算法开发者尝试更多新的想法，加快研发进度。TensorFlow 对分布式训练的支持比较完善。尤其是引入对 MPI、NCCL 等的支持之后，分布式效率有了很大提升。自 Google 提出 AutoML 以来，采用强化学习自动寻找最优的深度学习模型架构可能会成为未来的重要趋势。然而，基于强化学习的深度学习模型架

构优化会不断尝试不同的架构并实验，其消耗的计算资源会更多，这使得分布式训练更为必要。可以预见，以后模型或算法开发者对分布式训练的需求会越来越多。

- 深度学习模型的训练是一个非常考验调参技巧的过程。一些关键的参数包括梯度下降时的学习速率、批大小等。到现在为止，学术界和工业界一直没有找到很好的自动调参方法，仍需依赖人工经验。因此，在深度学习模型训练过程中，数据科学家或算法工程师的参与非常必要。这就要求深度学习引擎能够提供便利的工具链，包括可视化工具、调试工具、调优工具等。TensorFlow 提供的 TensorBoard 就是一款很好的可视化工具，方便我们观察训练过程中任意参数和梯度的变化，并及时终止超参设置不好的模型。在开源贡献者的努力下，其他 TensorFlow 周边工具也在不断涌现。
- 深度学习模型一旦训练完成、达到预期的精度之后，通常就需要被部署并提供推理服务。模型的训练一般在云侧进行，而推理可以在云侧或端侧进行。就人工智能应用的推理态而言，相比于云侧，目前端侧的市场空间可能更大，例如城市安防、自动驾驶等都是潜在的广阔应用领域。因此，深度学习引擎需要满足端侧轻量化部署的需求。TensorFlow 在设计之初也充分考虑了端侧的特点。XLA 特性支持为异构硬件生成目标代码，不仅可以提升模型运行效率，而且可以实现轻量化部署；TensorFlow Lite 也为移动设备上的模型运行提供了多种优化手段。

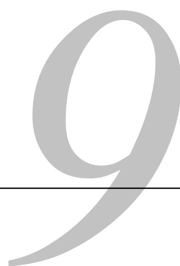
综上，深度学习对于张量计算性能、算子灵活性、自动微分能力、分布式训练、可视化和端侧部署等都有很强的诉求，而 TensorFlow 的设计也充分考虑到了这些因素，这使得它成为了当前业界很流行的一个深度学习引擎。此外，在 Google 研发团队和广大开源爱好者的努力下，TensorFlow 项目本身在飞速迭代和进步。例如，在易用性方面，TensorFlow 新近推出的 Eager 模式支持命令式编程，使得算法的开发、调试更加方便；Estimator、Dataset、Experiment 等高级 API 则简化了复杂模型的设计过程。TensorFlow 项目的 contrib 目录汇集了从模型、算法到系统等多个层面的大量第三方贡献，使得研发人员能够快速利用更多新的特性来加速深度学习理论研究和应用开发。可以看出，深度学习的理论、算法同深度学习计算引擎的发展总是相互影响的，理论和算法的突破会对计算引擎提出更多的需求，而计算引擎本身的增强也会有力地推动理论和算法的进步。

8.4 小结

深度学习源自 1957 年出现的感知机模型，在六十年左右的发展中经历了两次寒冬和两次复苏。最近几年，深度学习的爆发源于互联网和智能终端时代数据量的暴增、高性能和分布式计算设备处理能力的增强，以及人工智能与神经网络算法的改进等因素。由于深度神经网络模型在非线性函数拟合方面具有独特的优势，它在计算机视觉、自然语言处理等具有非结构化数据的领域取得了显著的应用效果，并成为了强化学习的助推剂。各类开源计算引擎也极大地促进了这一领域的发展，使得新想法得以快速实验，形成了算法与系统领域的有益互动与良性循环。TensorFlow 之所以能够在众多引擎中独占鳌头，归功于它在算子灵活性、分布式训练性能和工具链完善性等方面的优势，同时也离不开社区的贡献。

第 9 章

CNN 模型



人工智能希望赋予机器的能力包括感知、决策、推理和联想等。当前，深度学习最成熟的应用是感知，典型场景包括语音识别、图像识别等。CNN（卷积神经网络）模型对于非结构化数据具有出色的特征提取能力，是一种经典的感知模型，已被广泛应用于图像、语音和视频数据的处理与分析。本章首先分析 CNN 模型中的关键层，并介绍当前业界几种经典的 CNN 模型结构。然后介绍 TensorFlow 社区中用于图像分类的 Python 库——TensorFlow-Slim，并讲解其软件架构和使用流程。最后介绍 CNN 模型用于物体检测、图像分割等计算机视觉问题的实例，以帮助读者开拓应用思路。

9.1　CNN

神经网络和深度学习模型种类繁多，其中前馈神经网络是相对比较成熟、应用比较广泛的一类。前馈神经网络模型是指数据流只沿一个方向传递的模型。这类模型的主要特征表现为：(1) 通常由多层神经元构成；(2) 不同层的神经元之间存在连接，而同一层内的神经元之间没有连接。常见的前馈神经网络包括 CNN 和 DBN 等。相比于其他前馈神经网络，CNN 模型的参数较少，泛化能力强，它是当前深度学习领域的研究热点之一，其应用也越来越广泛。本节首先简要介绍 CNN 模型，然后重点介绍该模型包含的主要层以及几种经典模型。

9.1.1　CNN简介

CNN 模型是一种以卷积为核心的前馈神经网络模型。卷积是分析数学中的一种基础运算，其中对输入数据做运算时所用到的函数称为卷积核。卷积运算是指卷积核以滑动窗的形式在输入数据的各个位置上做小范围加权和的过程。以图像识别为例，卷积核在输入图像上不断滑动时，卷积核与当前滑动窗口内的输入图像像素值相乘后求和即得到输出图像的像素值。这个运算过程与图像处理算法中常用的空间滤波是类似的。因此，卷积可以被通俗地理解为一种“滤波”过程，卷积核与输入数据作用之后得到了“滤波”后的图像，从而提取出了图像的特征。

CNN 模型的基本架构如图 9-1 所示，可以看出，CNN 模型由一系列的层不断堆叠而成。一些复杂的 CNN 模型可以采用多个堆叠的层同时对输入数据做处理。卷积层是 CNN 模型的主要组成模块之一，它所对应的数学操作即为卷积运算。除了必备的卷积层和激活层之外，CNN 模型

还可以包含其他层，如池化层、全连接层、批归一化层等。CNN 模型的每一层都可以代表一种数学操作。这些操作有些是线性的，比如卷积、全连接、批归一化操作等；有些是非线性的，比如池化、激活操作等。非线性操作的存在保证了 CNN 模型是一个非线性模型，使其具备强大的学习能力。训练后的 CNN 模型整体上可看作一种非线性映射，例如，它可以将图像像素映射到该图像的类别，或者可以将音频信号映射到其相应的语义信息。

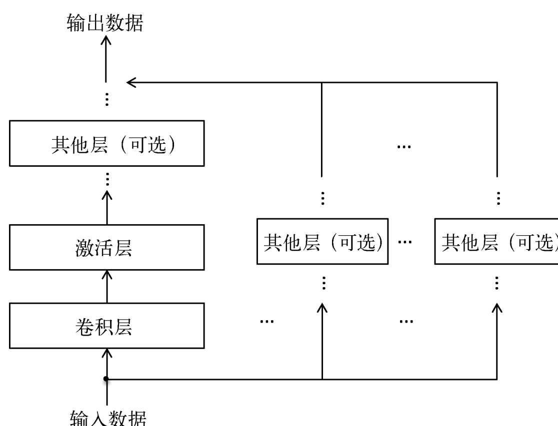


图 9-1 CNN 模型的架构图

9.1.2 卷积层

卷积层是使用一系列卷积核与多通道输入数据做卷积的线性计算层。卷积层的提出是为了利用输入数据（如图像）中特征的局域性和位置无关性来降低整个模型的参数量。如图 9-2 所示，假设该卷积层的输入数据为三张二维的特征图 $\boldsymbol{I}(1), \boldsymbol{I}(2)$ 和 $\boldsymbol{I}(3)$，输出数据为两张二维的特征图 $\boldsymbol{O}(1), \boldsymbol{O}(2)$（输入和输出通道数分别为 3 和 2）。该卷积层的参数由矩阵 $\boldsymbol{C}(i,j)$ 和 $b(i)$ 表示，其中，$i=1,2$ 对应输出数据的各个通道，$j=1,2,3$ 对应输入数据的各个通道。卷积层的整体计算过程为：当 $i=1$ 时，卷积核 $\boldsymbol{C}(1,j)$ 与二维数据 $\boldsymbol{I}(j)$ 分别执行卷积计算（其中 $j=1,2,3$），然后求和，再加上偏置 $b(1)$，得到第一个输出通道对应的输出 $\boldsymbol{O}(1)$。同理，当 $i=2$ 时，卷积核 $\boldsymbol{C}(2,j)$作用于二维数据 $\boldsymbol{I}(j)$，再加上偏置 $b(2)$，得到第二个输出通道对应的输出 $\boldsymbol{O}(2)$（其中 $j=1,2,3$）。由此可以看出，卷积层的计算是线性计算。

卷积计算的结果与卷积核大小 k（假设卷积核的长度和宽度相同）、步长 s 和补零 p 这三个参数密切相关。以卷积核 $\boldsymbol{C}(1,1)$ 与 $\boldsymbol{I}(1)$ 的卷积计算为例，具体计算方法如下。

(1) 将原始输入数据在 x 和 y 方向上补零。比如 $p=1$，则原始输入数据的四周多了一圈值为 0 的点。

(2) 在补零后的二维数据上，从左上角开始，大小为 $k\times k$ 的卷积核分别在 x 和 y 方向上以间隔 s 进行滑动，完成对二维数据的“扫描”。在卷积核滑动到的每个位置上，以该位置为中心的

面积为 $k \times k$ 的输入数据与该卷积核中的数据在对应位置上一一相乘后求和，就得到该位置所对应的输出值。假设输入数据的大小为 $m \times m$，则输出数据的长（或宽）为 $n = (m+2p-k)/s+1$。TensorFlow 提供了两种补零方式：VALID 和 SAME。如果是 VALID 方式，则 $p = 0$，且$(m-k)/s$计算结果的小数部分会被舍弃（即向下取整），所以 $n = \lfloor (m-k)/s \rfloor + 1$。如果是 SAME 方式，则输入数据的四周均匀补零，补零的总数为$\lceil (m-k)/s \rceil \times s + k - m$，输出数据的长（或宽）为 $n = \lceil (m-k)/s \rceil + 1$（其中，$\lfloor x \rfloor$表示对 x 向下取整，$\lceil x \rceil$表示对 x 向上取整。比如当 $x = 5.3$ 时，$\lfloor x \rfloor = 5$，$\lceil x \rceil = 6$）。

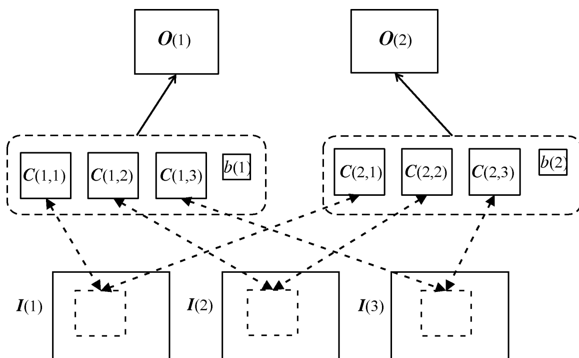


图 9-2　CNN 模型中的卷积计算示意图

除了上述卷积计算之外，也有其他形式的卷积计算，比如扩张卷积（Dilated Convolution）。在扩张卷积中，当卷积核与对应位置的滑动窗口内输入的数据相乘时，卷积计算还受另外一个参数 d 的控制。图 9-3a 为传统卷积计算。对于某个输入通道，卷积核大小为 3×3，输入数据大小为 5×5。假设卷积核滑动到输入数据的最中心点并覆盖到 3×3 的窗口，则该点的卷积输出值为卷积核与该窗口内对应点的乘积和。图 9-3b 为扩张卷积计算（$d = 1$）。当卷积核滑动到输入数据的中心点时，卷积核“所覆盖的窗口”大小为 5×5，卷积核与该窗口内带有纹理的像素点相乘后求和输出即为该点的扩张卷积输出值。因此，扩张卷积运算使得卷积核覆盖的区域更广，能够将局域特征和非局域特征融合，从而提取多尺度的特征信息。

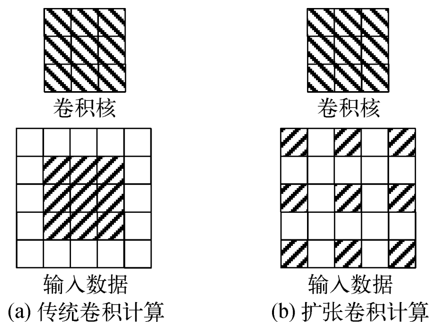


图 9-3　传统卷积计算和扩张卷积计算对比

9.1.3 激活层

激活层是使用激活函数对输入数据做处理的一种非线性层。卷积等计算本质上是对输入数据的线性变换，为了实现神经网络的非线性建模能力，激活层都采用非线性函数。常用的激活函数有 sigmoid、tanh、ReLU 等，它们的特性对比如表 9-1 所示。

表 9-1 常用的几种激活函数特性对比

激活函数	公式	输出范围	优 点	缺 点
sigmoid	$y=\frac{1}{1+e^{-x}}$	[0,1]	可直接用在输出层	(1) 神经元输出非 0 均值，不易于模型训练；(2) 容易饱和，造成后向传播时“梯度消失”；(3) 前向和后向计算复杂
tanh	$y=\frac{e^x-e^{-x}}{e^x+e^{-x}}$	[−1,1]	神经元输出为 0 均值，易于模型训练	(1) 容易饱和，造成后向传播时“梯度消失”；(2) 计算复杂
ReLU	$y=\max(0,x)$	[0,+∞)	(1) 具有单侧抑制性，神经元兴奋域宽，激活具有稀疏性，与脑神经元激活频率函数具有相似之处；(2) 梯度不容易饱和；(3) 计算和求导速度快	容易出现参数为负、梯度为 0 的情况而导致神经元“坏死”，无法再激活

除上述几种常用的激活函数之外，还有其他激活函数。比如为了避免 ReLU 激活函数的缺点，有人提出了 Parametric ReLU 激活函数。它与 ReLU 的区别在于：当输入为负数时，输出与输入仍然成线性关系，但是线性比例值可以在训练过程中不断调整。这使得输入为负数时，神经元也可以被激活。

9.1.4 池化层

池化层是用于缩小数据规模的一种非线性计算层。为了降低特征维度，我们需要对输入数据进行采样，具体做法是在一个或者多个卷积层后增加一个池化层。池化层由以下三个参数决定：

- 池化类型，一般有最大池化和平均池化两种；
- 池化核的大小 k；
- 池化核的滑动间隔 s。

图 9-4 给出了一种 $k=2, s=2$ 的池化层示例。其中，2×2 大小的池化窗口以两个单位距离在输入数据上滑动。在池化层中，如果采用最大池化类型，则输出为输入窗口内四个值的最大值；如果采用平均池化类型，则输出为输入窗口内四个值的平均值。

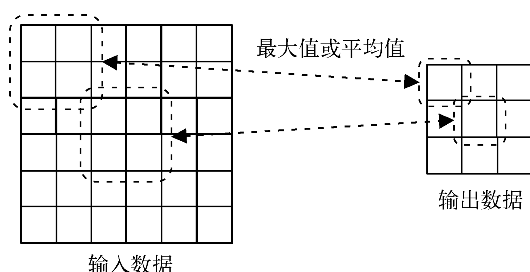


图 9-4　池化层示意图

9.1.5　全连接层

全连接层是一种对输入数据直接做线性变换的线性计算层。它是神经网络中最常用的一种层，用于学习输出数据和输入数据之间的线性变换关系。全连接层可作为特征提取层使用，在学习特征的同时实现特征融合；也可作为最终的分类层使用，其输出神经元的值代表每个输出类别的概率。基于 CNN 的分类模型的最后一层或者几层都采用全连接层。如图 9-5 所示，每个输出神经元都与每个输入神经元相连接，并且连接权重可能不相同。输出神经元的值为输入神经元值的加权和。在 CNN 中，全连接层与卷积层相比，计算速度比较快。但是，全连接层内部是稠密连接，使得参数量很大，这会导致分布式训练时通信开销比较大。CNN 模型的前面几层之所以不用全连接层而用卷积层，很重要的一个原因就是全连接层会因参数太多而造成无法训练。

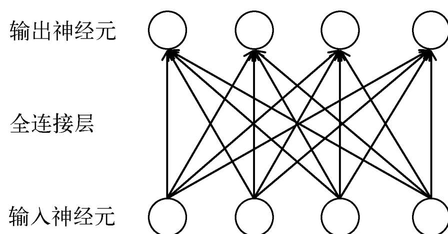


图 9-5　全连接层示意图

相比于卷积层，全连接层的限制在于它的输入神经元个数是固定的。如果我们使用带有全连接层的 CNN 模型，那么其输入数据的尺寸必须是固定的。CNN 模型并非一定需要全连接层，比如在用于图像分割任务的 FCN（Fully Convolutional Network，全卷积网络）中，就没有全连接层。

9.1.6　Dropout层

Dropout 层是一种正则化层。正是因为全连接层参数量非常庞大（占据了 CNN 模型参数量的 80% ~ 90%），发生过拟合问题的风险比较高，所以我们通常需要一些正则化方法训练带有全连接层的 CNN 模型。Dropout 是常用的一种正则化方法。如图 9-6 所示，在每次迭代训练时，将神经元以一定的概率值 p 暂时随机丢弃，即在当前迭代中不参与训练。这样，每次训练时的模型都由

不同的参数组合而成。神经元的随机组合减少了神经元之间可能形成的共同依赖。Dropout 方法使得最终训练的神经网络由多个 Dropout 之后的子模型组成。最后预测的时候，所有神经元都一起参与计算，相当于集成了之前训练的所有子模型，这有助于提升模型的泛化能力。

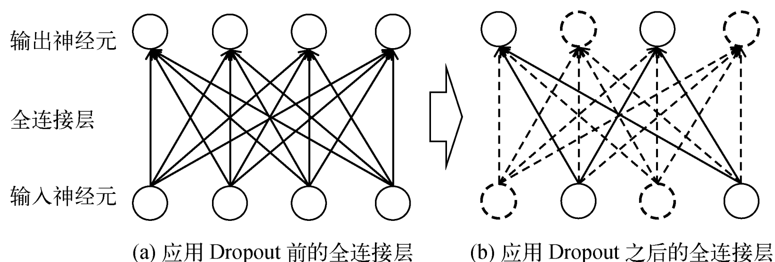


图 9-6 Dropout 方法示意图

9.1.7 BN层

在训练 CNN 模型时，模型参数随迭代次数的增加而不断变化。对于每一层而言，其输入数据的分布也随迭代次数的不同而不同。因此，模型参数要不断去适应这种随迭代变化的输入分布，这会导致模型参数学习很慢。如果能使得每个卷积层输入的分布固定——比如固定为均值为 0、方差为 1 的高斯分布，那么模型参数的学习将会更容易。这种将每层输入的分布做归一化的操作叫作批归一化，相应的层为 BN（Batch Normalization，批归一化或批标准化）层。

一般而言，CNN 模型在卷积或全连接运算完成之后进行 BN 运算，然后再激活。对于卷积层，BN 层的具体计算过程为：假设某输出特征图为 $\boldsymbol{O}(i, j, h, w)$，其中 $i = 1, 2, \cdots, N$ 为该样本在本次迭代中所用到的训练样本集合（mini-batch）中的索引号，$j = 1, 2,\cdots, C$ 为该特征图所对应的通道索引号。对于通道 j，我们可以求得在 mini-batch 维度上所有特征图的平均值 $\boldsymbol{\mu}(j) = \frac{1}{NHW}\sum_{n=1}^{N}\sum_{h=1}^{H}\sum_{w=1}^{W}\boldsymbol{O}(i,j,h,w)$ 和方差值 $\boldsymbol{\sigma}(j) = \frac{1}{NHW}\sum_{n=1}^{N}\sum_{h=1}^{H}\sum_{w=1}^{W}(\boldsymbol{O}(i,j,h,w)-\boldsymbol{\mu}(j))^2$，然后利用 $\boldsymbol{\mu}(j)$ 和 $\boldsymbol{\sigma}(j)$ 对每个特征图上的特征点进行归一化，并将归一化后的值再进行线性变换（变换系数通过训练得到），得到 BN 层最后的输出值。全连接层的 BN 运算也是类似的，只要将上述 H 和 W 的维度变为 1 即可。

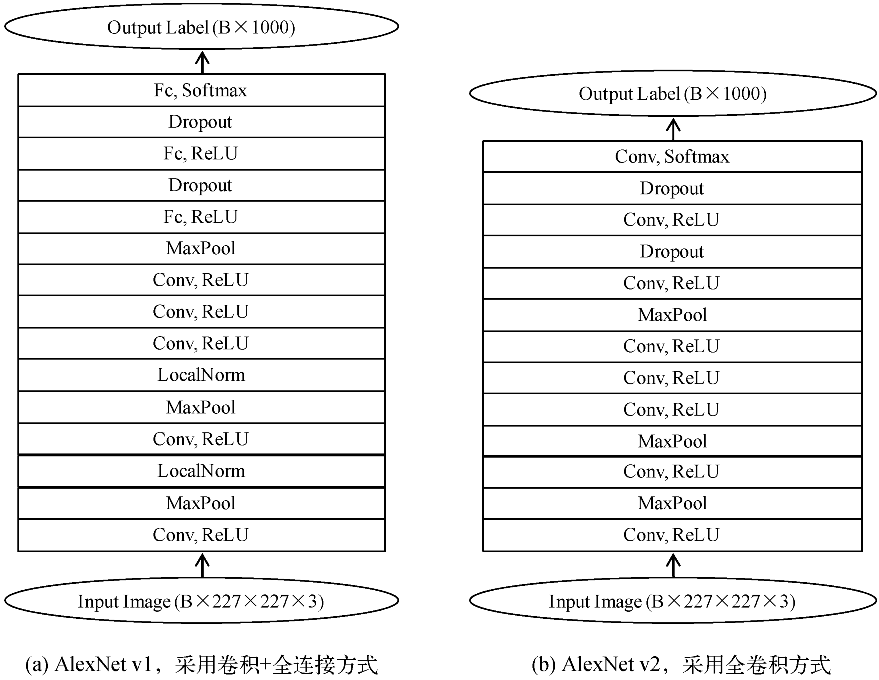
加入 BN 层之后，训练数据在样本空间中的分布更加均匀和固定，所以每个卷积或全连接层所形成的高维切面对于数据的切分更加容易，这使得模型参数的学习也变得更容易。因此，BN 层可以显著加速 CNN 模型训练的收敛速度。例如，在使用 BN 层之后，GoogleNet（Google 提出的一种经典模型）的训练速度提升了 10 倍以上。

9.1.8 常用的CNN图像分类模型

CNN 模型最先应用于图像识别。这类模型通常由很多层堆叠而成，例如多个连续堆叠的卷

积层和激活层构成的网络可以学习由低级到高级逐渐变化的层次化图像特征。AlexNet、VGG、Inception、ResNet 等都是经典的 CNN 模型，它们各自都有不同的版本。

AlexNet 最先在 ImageNet 图像分类竞赛中获得突破，其模型架构如图 9-7 所示。原始的 AlexNet 模型（AlexNet v1，参见图 9-7a）主要由 5 个卷积层和 3 个全连接层组成。TensorFlow-Slim（一个用于 CNN 模型训练和推理的 Python 库）还提供了 v2 版本的 AlexNet，如图 9-7b 所示。AlexNet v2 去掉了 LocalNorm 层（现在的 CNN 模型基本上都不加这一层），并在最后几层中使用卷积层代替全连接层，但是模型的计算量和参数量基本没有变化。



(a) AlexNet v1，采用卷积+全连接方式　　(b) AlexNet v2，采用全卷积方式

图 9-7　AlexNet 模型架构

牛津大学的研究人员于 2014 年提出了 VGG 模型，旨在提供比 AlexNet 更深的模型以提升图像分类精度。他们发现大量采用 3×3 卷积核的卷积层已经能够很好地提取图像特征。VGG 模型分很多种，常用的是 VGG16 和 VGG19 模型，分别如图 9-8a 和图 9-8b 所示。VGG 模型在多层卷积之后采用了三个连续的全连接层来融合多通道特征图，这使得模型参数量增多、模型变大。例如，VGG16 模型的规模在 500MB 以上，这给分布式训练和端侧部署都带来不小的挑战。

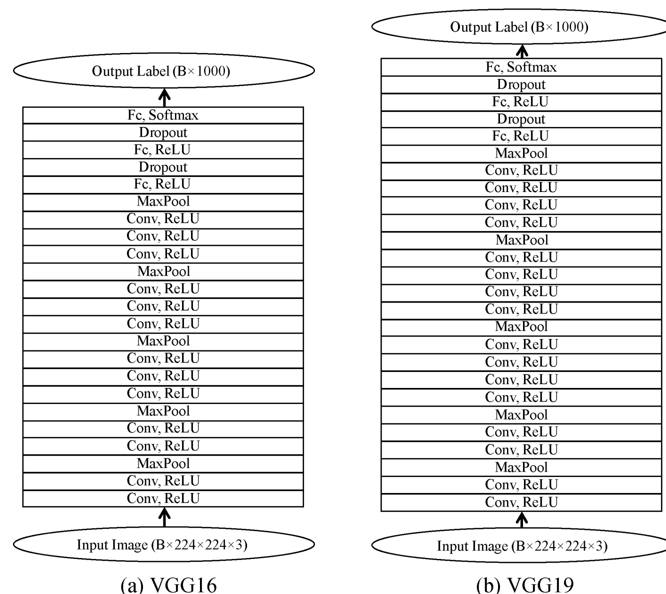


图 9-8 VGG 模型架构

为了更好地融合多尺度模型特征，同时避免采用过多的全连接层，Google 公司的研究团队提出了 Inception 模型及其变种。Inception 系列模型一共包括四种版本，分别如图 9-9a、图 9-10a、图 9-11a 和图 9-12a 所示。Inception v1 采用 Inception module1（详见图 9-9b）提取并融合不同尺度下的特征，并且在 3×3 和 5×5 卷积核之前采用 1×1 的卷积核降维以减少计算量。Inception v2 则改用 Inception module2（详见图 9-10b）。与 Inception module1 相比，Inception module2 将 5×5 卷积变为两个 3×3 卷积，降低了计算量。

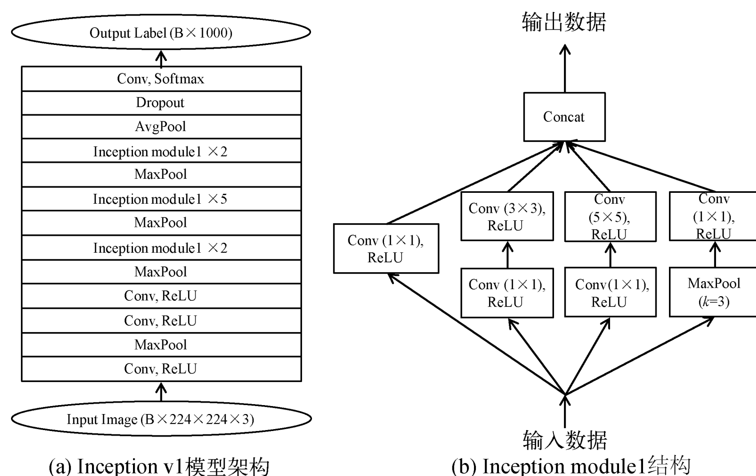


图 9-9 Inception v1 模型架构及其关键模块

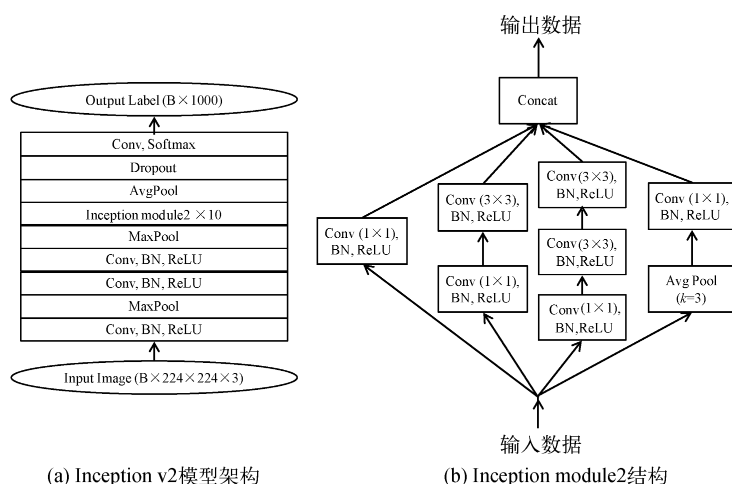


图 9-10　Inception v2 模型架构及其关键模块

Inception v3 采用与 Inception module2 相似的 Inception module3 作为初始的 Inception 模块。Inception module3 的输出特征经 Inception module4 降维后，再经过 Inception module5 做进一步特征抽取和融合。Inception module5 将 $n \times n$ 的卷积运算转换为 $1 \times n$ 和 $n \times 1$ 的连续卷积，可以进一步降低计算量。Inception module6 与 Inception module4 类似，也具有降维作用。Inception module7 与 Inception module5 类似，但是可以融合更多通道的卷积结果，提升高维的特征表达能力，一般用于 CNN 的最后几层中。Inception v4 相比 Inception v3，网络深度更深。在 Inception v4 中，Inception module8 用于模型的初期特征提取和多尺度融合。与 Inception v1 和 Inception v2 不同，Inception v3 和 Inception v4 均要求输入的图像大小为 299×299。

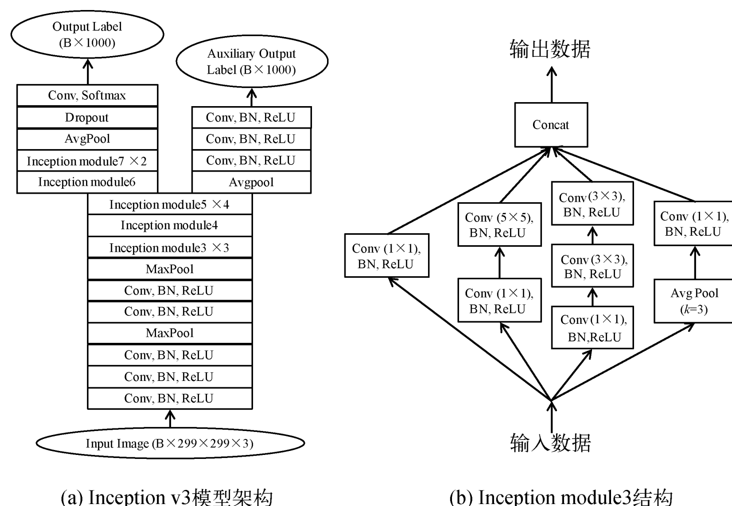
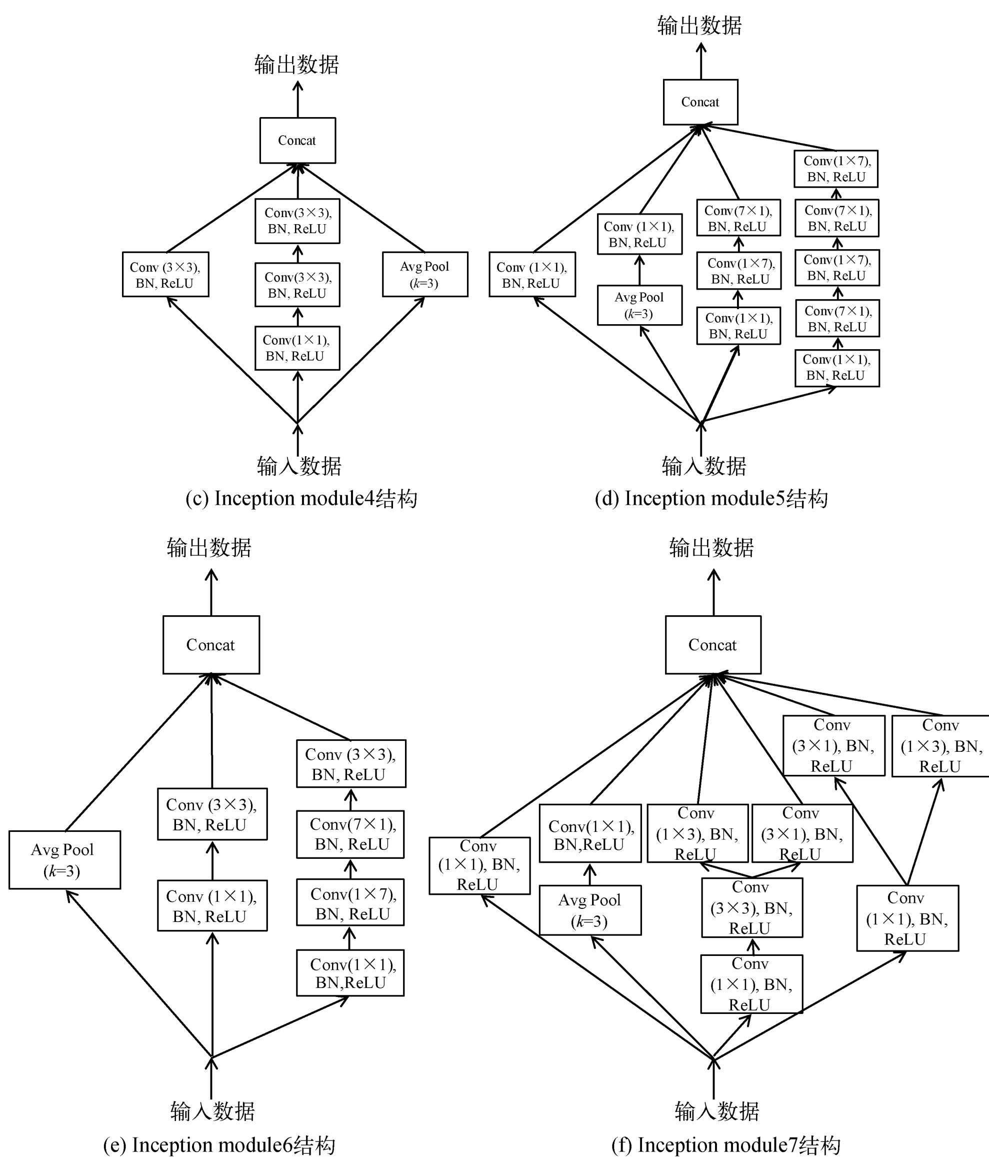


图 9-11　Inception v3 模型架构及其关键模块



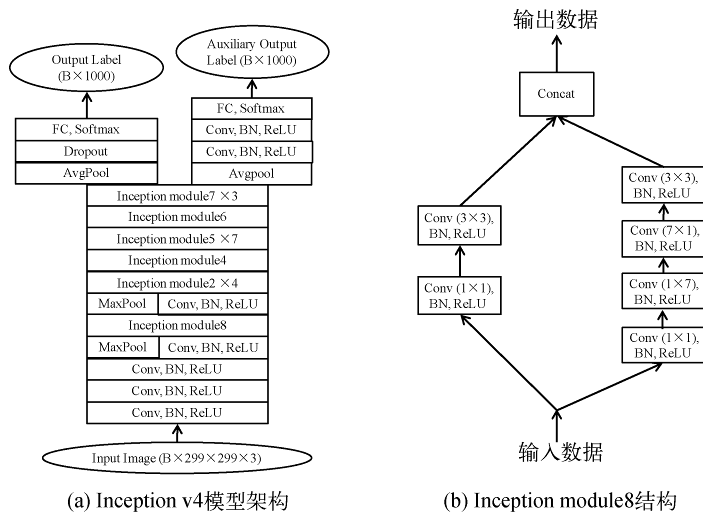
(c) Inception module4结构

(d) Inception module5结构

(e) Inception module6结构

(f) Inception module7结构

图 9-11 （续）

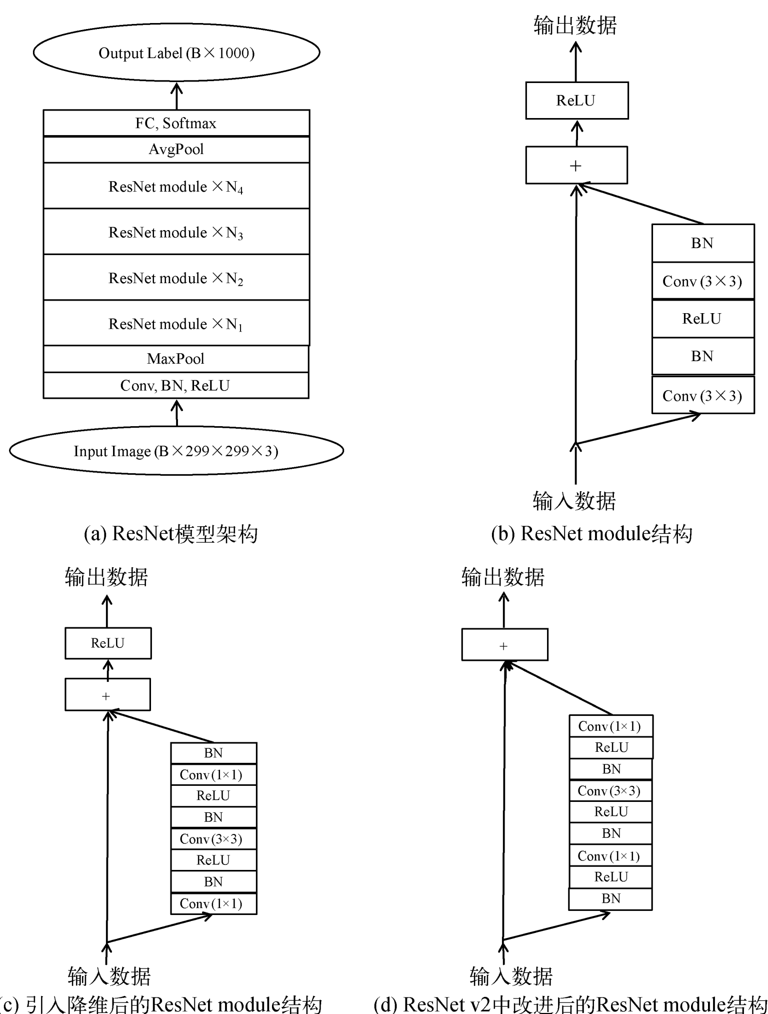


(a) Inception v4模型架构　　(b) Inception module8结构

图 9-12　Inception v4 模型架构及其关键模块

从 AlexNet 到 VGG 和 Inception 模型，模型深度一直在增长，精度也在不断提高。但是当层数继续增加的时候，训练误差和验证误差反而都会增加，精度开始下降。其问题不是过拟合，而是难以找到更好的训练更深层网络的方法。为了解决这个问题，2015 年微软亚洲研究院的何恺明等人提出了 ResNet。图 9-13a 给出了该模型的架构。其中，输入数据经过一层卷积之后通过 4 组 ResNet module 处理（每组分别包含有 N_1, N_2, N_3, N_4 个 ResNet module），每经过一组 module 后特征图分辨率降低一次。ResNet 的一个基本假设是：直接使用多个卷积层学习一个非线性映射比较困难，但是让其学习输入和输出之间的残差则比较容易。ResNet module 用于学习这个残差，每个 ResNet module 的输出数据都由输入数据和残差共同构成。图 9-13b、图 9-13c 和图 9-13d 展示了三种不同的 ResNet module。由于 module 中的输入和输出之间出现了“直连通道”，梯度传播会更容易，从而有利于构建更深的模型。大量的实验证明了这种学习方式的有效性。ResNet 模型可以达到的深度也远超过前面提到的 AlexNet、VGG 和 Inception 模型。

根据层数的不同，ResNet 有不同的版本，其中常用的有 ResNet-18、ResNet-34、ResNet-50、ResNet-101 和 ResNet-152。对于 ResNet-18 和 ResNet-34，其 ResNet module 如图 9-13b 所示，中间包含 2 个 3×3 的卷积运算，4 组 ResNet module 中卷积计算的输出通道数依次为 64、128、256、512。ResNet-50、ResNet-101 和 ResNet-152 中的 ResNet module 如图 9-13c 所示。为了减少计算量，该 ResNet module 先用 1×1 的卷积降维（通道数降低）后再做 3×3 卷积，最后用 1×1 的卷积提升特征通道数。在 ResNet v2 中，ResNet module 如图 9-13d 所示。为了使梯度从深层到浅层传播过程中不发生梯度弥散和梯度爆炸现象，该 ResNet module 的主线路上不添加 ReLU 模块。



(a) ResNet模型架构　(b) ResNet module结构

(c) 引入降维后的ResNet module结构　(d) ResNet v2中改进后的ResNet module结构

图 9-13　ResNet 模型架构及其关键模块

在 Inception 模型和 ResNet 模型之后，Google 又提出 Inception-ResNet 系列模型，将 ResNet 中的残差模块（ResNet module）设计和 Inception 架构相融合以充分利用彼此优点。表 9-2 中总结了以上几种常用的 CNN 分类模型。

表 9-2　常用的分类模型列表

网络名称	主要设计思想	计算复杂度	模型大小(MB)
AlexNet	首次采用多层深度 CNN 模型做图像识别；采用 ReLU 避免梯度消失；提出 Dropout 技术防止模型过拟合	低	≈240
VGG	为了降低计算量，全部采用 3×3 大小的卷积核，通过连续的 3×3 卷积模拟 5×5 或 7×7 等更大的卷积核	高	≈550

（续）

网络名称	主要设计思想	计算复杂度	模型大小（MB）
Inception（v1~v4）	为了融合多种感受野大小下的特征图，提出了 Inception 模块，该模块包含了 1×1、3×3 和 5×5 的卷积层，并在 3×3 和 5×5 卷积前采用 1×1 卷积降低特征通道数，减少计算量	中	50~163
ResNet	提出了残差模块，通过用多层卷积去学习输入和输出之间的残差，而非直接学习输入和输出之间的映射，使得深层网络更容易训练。当深度大于 50 之后，在 3×3 卷积运算前采用 1×1 降低计算量	较高	90~214
Inception-ResNet	在 Inception 模块中引入输入和输出之间的直连通路，即采用残差训练方法，将残差模块的设计思想融入到 Inception 架构中	较高	≈200

除了上面介绍的模型外，近年来一些旨在提升计算效率和精度的新兴 CNN 分类模型也不断涌现，例如 ResNeXt、DenseNet、DPN（Dual Path Network）等。在提升精度的同时，这些模型也在考虑尽可能降低计算量。另外，现在业界也倾向于采用强化学习、进化算法等方法自动探索和设计最优的 CNN 分类模型架构。CNN 分类模型中除去分类层（Softmax 层）以外的部分通常作为通用的特征提取器，可用于计算机视觉领域中的其他任务（如检测、分割等）。

9.2　TensorFlow-Slim

上节讲述了 CNN 中常见的层和典型的模型结构，但这仅停留在理论层面，我们需要具体实践才能更深入地理解和运用这一模型。同时，对于一个实际的应用，仅有模型本身是不够的，我们还需要数据读取和预处理、模型部署管理、优化器和损失函数（用于训练）、评估函数（用于推理）等模块。TensorFlow-Slim 作为一套深度学习高级抽象库，旨在简化上述程序开发流程。本节讲解如何使用 TensorFlow-Slim 实现 CNN 模型从训练到推理的完整流程。

9.2.1　TensorFlow-Slim总体结构

由于 CNN 等深度学习模型往往非常复杂，使用 TensorFlow 原生 API 编写其训练和推理逻辑的代码量会很大。TensorFlow-Slim 库的推出正是为了方便用户快速定义和训练自己的模型，并利用训练好的模型做推理。TensorFlow-Slim 是一个 Python 库，它提供了包括数据集处理、模型定义、模型训练和推理在内的诸多功能。需要注意的是，虽然 TensorFlow-Slim 最初面向的是图像分类应用，但是我们也可以将这套框架应用于其他模型和数据集，从而解决其他问题。比如，我们可以自定义一个 CNN 和 RNN 混合的模型用于光学字符识别（Optical Character Recognition，OCR），然后复用 TensorFlow-Slim 框架做模型训练和推理。

TensorFlow-Slim 的代码托管于 TensorFlow 社区 models 项目的 research/slim 目录，以及 tensorflow 项目的 tensorflow/contrib/slim 目录。如图 9-14 所示，TensorFlow-Slim 的主要组件包括 `datasets`、`data`、`preprocessing`、`deployment` 和 `nets` 等 Python 包（大部分位于 models 项目的 research/slim 目录下），以及 `learning`、`evaluation`、`queues` 和 `model_analyzer` 等 Python

模块（位于 tensorflow 项目的 tensorflow/contrib/slim 目录下）。这些组件分别提供了数据读取和处理、模型定义、模型训练和模型推理等功能。TensorFlow-Slim 依赖的其他 Python 包主要有 `layers`、`ops`、`losses`、`metircs` 等。这些组件定义了模型用到的层、算子、损失函数和度量指标等。下面我们分别介绍 TensorFlow-Slim 中的几个主要组件。

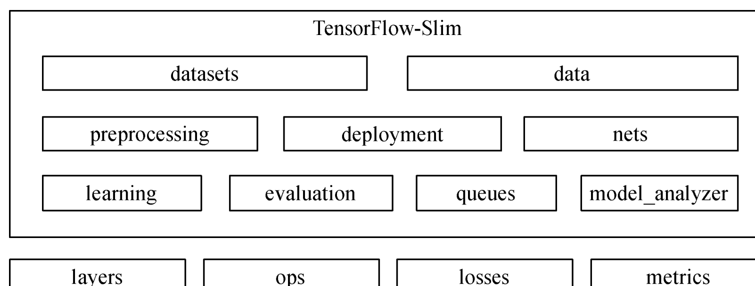


图 9-14　TensorFlow-Slim 的主要组件及其依赖组件

9.2.2　datasets包和data包

`datasets` 包提供了读取输入数据的统一接口，它支持 ImageNet、MNIST、CIFAR-10、Flowers 这四种标准数据集的读取。因为每个数据集的元信息以及读取方式有所不同，所以 `datasets` 包针对每个数据集提供了一个对应的模块进行处理。用户也可以在 `datasets` 包中添加模块以支持新的数据集。

`dataset_factory` 模块用于管理目前支持的数据集，并提供统一的获取数据的接口。该模块的 `get_dataset` 方法根据用户输入的数据集名 `name`、数据类型名 `split_name`（取值为 `train` 或 `test`）、所在目录 `dataset_dir`、数据集文件命名规则 `file_pattern` 和数据集读取器 `reader`，调用相应数据集的 `get_split` 方法，然后返回一个 TensorFlow-Slim 所定义的数据集对象：

```
from datasets import cifar10
from datasets import flowers
from datasets import imagenet
from datasets import mnist
# 数据集 map，当前支持以下这四种，每种数据集都对应一个单独的模块，都有 get_split 方法
datasets_map = {
    'cifar10': cifar10,
    'flowers': flowers,
    'imagenet': imagenet,
    'mnist': mnist,
}

def get_dataset(name, split_name, dataset_dir, file_pattern=None, reader=None):
  if name not in datasets_map:
    raise ValueError('Name of dataset unknown %s' % name)
  return datasets_map[name].get_split(
      split_name,
```

9

```
        dataset_dir,
        file_pattern,
        reader)
```

data 包用于数据的多线程读取、解码等，并为 datasets 包提供封装好的 Dataset 类。其代码位于 tensorflow/contrib/slim/python/slim/data 目录下。data 包中主要组件所对应的 UML 类图如图 9-15 所示。

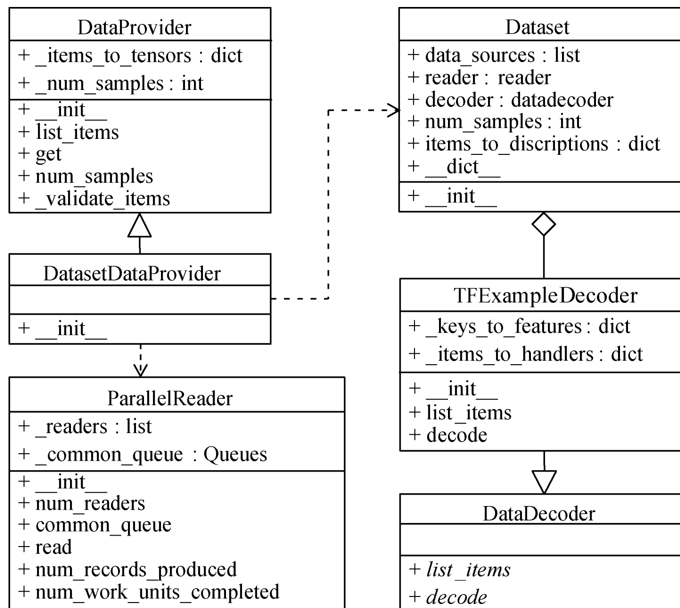


图 9-15 data 包主要组件的 UML 类图

其中，DataProvider 类维护了输入数据包含的关键字（在代码中用 item 表示）及其对应张量。比如，对于 ImageNet 分类问题，关键字为 image 和 label，对应的张量分别用于存放图像数据和图像标签。DatasetDataProvider 类继承于 DataProvider 类。在 DatasetDataProvider 对象初始化时，ParallelReader 类的 parallel_read 方法被调用，以读取 dataset.data_sources 指定的数据源，得到序列化的数据。然后，TFExampleDecoder 类的 decode 方法被调用，对该数据进行反序列化。

tfexample_decoder 模块还定义了 ItemHandler 类及其子类，如 BoundingBox、Image、Tensor、SparseTensor 和 ItemHandlerCallback。这些类主要维护了一些关键字及其对应的序列化数据，相关的 UML 类图如图 9-16 所示。这些类在 TFExampleDecoder 的 decode 方法中会被用到。以 Image 类为例，该类维护了两个重要的关键字—— _image_key 和 _format_key，他们分别对应序列化之后的图像像素数据 image_buffer 和图像格式字符串 image_format（如 jpg、png 等）。此外，Image 类还提供了 _decode 方法，用于解压 image_buffer 和 image_format 中的数据。

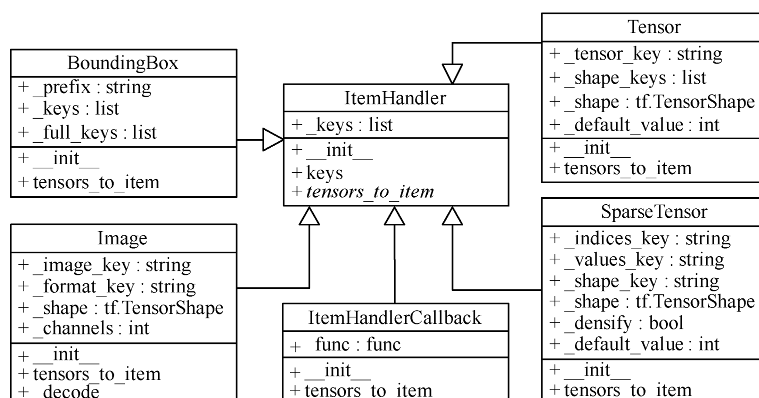


图 9-16 `ItemHandler` 及其子类的 UML 类图（仅画出主要的类）

在实际使用中，用户首先根据数据集名、数据类型名、数据集目录等参数新建 `Dataset` 对象，然后再建立一个 `DatasetDataProvider` 对象，用一个或多个 `Reader` 对象读取训练数据，并用 `Decoder` 对象将读到的数据解码为可被训练代码处理的张量（比如图像数据及其标签）。在训练过程中，我们通常需要将数据读取和模型计算并行化。在计算当前迭代的时候，下一次迭代的数据已经开始被加载到内存中，以保证数据读取等 IO 操作不会造成瓶颈。`data` 包中的 `prefetch_queue` 模块提供数据预取功能，该模块的 `prefetch_queue` 方法用于建立一个先进先出队列（FIFOQueue）。该队列可看作一个缓冲区，它以批大小为单位不断准备后续将训练的数据。

9.2.3 preprocessing包

该包提供多种数据预处理函数。在深度学习模型用于训练和推理时，我们通常需要对输入数据（如图像等）进行预处理，比如调整图像输入尺寸、截取部分图像等。`preprocessing` 包提供了对 CIFARNet、Inception、LeNet、VGG 这四类模型输入数据的预处理功能。以 Inception 类模型为例，`inception_preprocessing` 模块提供了用于图像色彩调整的 `distort_color` 方法，该方法可以以不同顺序分别调节图像的色调、饱和度、亮度和对比度。此外，该模块还提供了用于图像候选框调整（主要用于物体检测应用）的 `distorted_bounding_box_crop` 方法，可以将人工在图像中标注的候选框在一定尺寸、长宽比和重叠度范围内做畸变。

9.2.4 deployment包

该包提供用于模型训练态和推理态部署的方法库，具有 `model_deploy` 模块，该模块定义了三个主要的数据结构：`Clone`、`DeploymentConfig` 和 `DeployedModel`。

当我们用多个 GPU（或 CPU）等计算设备做数据并行训练时，每个计算设备都被部署一套相同的模型（或数据流图）。`model_deploy` 模块抽象出一个 `namedtuple` 类型的数据结构——`Clone`，使得每个计算设备上的模型对应一个 `Clone` 对象。`Clone` 的定义为：

```
Clone = collections.namedtuple('Clone',
                               ['outputs', # 该 Clone 所对应的深度学习模型的输出张量
                                'scope',   # 该 Clone 的作用域
                                'device',  # 该 Clone 的设备
                               ])
```

它包含三个属性：outputs、scope 和 device。其中，outputs 表示 CNN 模型 Softmax 层（或其他类型的输出层）的输出张量，它代表模型的预测值；scope 表示定义该 Clone 的作用域名称，格式为 clone_i，其中 i（$i=0,1,\cdots,N-1$）是当前节点内 GPU（或 CPU）的编号，N 为当前节点内计算设备的个数；device 表示该模型训练时所在设备的类型，当前 TensorFlow 仅支持 GPU 和 CPU 两种类型的设备。

DeploymentConfig 用于对模型训练态部署时的参数做配置，这些参数包括：Clone 对象的个数（num_clones）、是否在 CPU 上部署模型（clone_on_cpu）、计算节点上模型副本的 ID（replica_id）、计算集群中的模型副本总数（num_replicas）、参数服务器的数量（num_ps_tasks）、参数服务器所用的设备（ps_device），以及计算节点所用的设备（worker_device）。model_deploy 模块提供了 create_clones 方法，该方法根据一个 DeploymentConfig 对象和模型函数的定义创建指定数目的 Clone 对象。

DeployedModel 是一个更高层次的抽象，代表一个部署好的模型，可以帮助我们管理节点内的多个 Clone 对象。它也是一个 namedtuple 类型，其定义为：

```
DeployedModel = collections.namedtuple('DeployedModel',
                                       ['train_op',    # 单步训练操作
                                        'summary_op',  # 记录变量变化的操作
                                        'total_loss',  # 所有 Clone 对象上模型损失值的总和
                                        'clones',  # 多个 Clone 对象的集合
                                       ])
```

DeployedModel 类有四个属性：train_op、summary_op、total_loss 和 clones。其中，train_op 表示单步训练操作，summary_op 是用于记录变量变化的操作。在模型训练时，summary_op 能够记录模型参数、梯度和损失值等，以便使用 TensorBoard 查看。total_loss 表示所有 Clone 对象对应的模型损失值（包括正则项损失值在内）的总和。clones 表示一个 DeployedModel 对象所管理的多个 Clone 对象的集合。

model_deploy 模块的核心方法为 deploy。它的主要输入参数为 DeploymentConfig 对象、模型函数 model_fn，以及其他可选参数（如优化器 optimizer 等）。该方法用于计算多个 Clone 对象各自的单步训练操作、损失值等，并给用户提供一个 DeployedModel 对象。首先，该方法根据所需部署的 Clone 个数，在每个设备上建立一个 Clone 对象。如果输入参数已指定优化器 optimizer，则 global_step 被创建，用于表示所有参与计算的 worker 的同步标记（详见 5.2 节）。随后，通过优化器得到模型参数的梯度，并将所有梯度更新应用到相应的模型参数上。最后，得到当前迭代中所有更新操作聚合后的操作 train_op。相应代码为：

```
if optimizer:
  with tf.device(config.variables_device()):
```

```
    # 创建多个worker并行计算时的同步标记global_step
    global_step = slim.get_or_create_global_step()
    # 调用model_deploy模块中的optimize_clones方法，得到所有Clone的损失函数值的综合
    # total_loss和clones_gradients。clones_gradients的数据格式为:
    # [(grad_1,var_1),(grad_2,var_2),...,
    # (grad_i,var_i),...,(grad_N,var_N)]，其中var_i表示第i层的参数（weights或biases等），
    # grad_i表示var_i所对应的梯度，grad_i是所有Clone上计算得到的梯度的总和
    total_loss, clones_gradients = optimize_clones(clones, optimizer)
    if clones_gradients:
      if summarize_gradients:
        summaries |= set(_add_gradients_summaries(clones_gradients))
      # 调用优化器的apply_gradients方法，得到grad_updates操作，
      # 该操作利用clones_gradients中的梯度更新对应的参数
      grad_updates = optimizer.apply_gradients(clones_gradients,
                                              global_step=global_step)
      update_ops.append(grad_updates)
      update_op = tf.group(*update_ops)
      # 最终得到train_op，它表示完成一次迭代所需的所有更新操作
      train_op = control_flow_ops.with_dependencies([update_op], total_loss,
                                                    name='train_op')
```

如果 `deploy` 方法的输入参数中 `optimizer` 为 `None`，那么 `train_op` 为 `None`，通过计算所有 `Clone` 上损失值的平均值得到最终损失值 `total_loss`：

```
else:
  clones_losses = []
  regularization_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
  for clone in clones:
    with tf.name_scope(clone.scope):
      # 调用model_deploy模块中的_gather_clone_loss方法，得到当前Clone所对应的所有损失值，
      # 该损失值由tf.GraphKeys.LOSSES指定的损失值和输入参数regularization_losses
      # 指定的损失值共同构成。其中，tf.GraphKeys.LOSSES指定的损失值需要除以Clone的个数
      clone_loss = _gather_clone_loss(clone, len(clones), regularization_losses)
      if clone_loss is not None:
        clones_losses.append(clone_loss)
      # 除了第一个Clone之外，其他的Clone所对应的regularization_losses都被忽略
      regularization_losses = None
  if clones_losses:
    # 将所有Clone的损失值clones_losses相加，得到总的损失值total_loss
    total_loss = tf.add_n(clones_losses, name='total_loss')
```

此外，`deploy` 方法还指定了一些 `summary_op`，用于记录模型参数的梯度和损失值的变化。最终，`deploy` 方法返回一个 `DeployedModel` 对象。

9.2.5 nets包

该包提供多种CNN模型的实现，其中内置了很多模块，如 `alexnet`、`inception`、`vgg` 等。这些模块各自定义了一个CNN模型，例如alexnet.py定义了AlexNet v2模型。除此之外，`nets` 包还有一个 `nets_factory` 模块。该模块定义了两个字典——`networks_map` 和 `arg_scopes_map`，前者保存 `nets` 包中模型的名字与模型的实现函数之间的映射关系，后者保存 `nets` 包中模型的

名字与参数作用域（arg_scope）之间的映射关系。此外，该模块还定义了 get_network_fn 方法，用户可以调用该方法获得所需要的模型的实现函数。该方法的实现如下：

```
def get_network_fn(name, num_classes, weight_decay=0.0, is_training=False):
  if name not in networks_map:
    raise ValueError('Name of network unknown %s' % name)
  arg_scope = arg_scopes_map[name](weight_decay=weight_decay)
  # 在 network_map 字典中，通过指定的模型名字（如 alexnet_v2）
  # 得到一个模型的实现函数（如 alexnet.alexnet_v2 方法）
  func = networks_map[name]
  # 此处使用 functools.wraps 装饰器，在原函数 func 之上添加参数作用域 arg_scope，
  # 以避免在模型定义时重复写过多的参数
  @functools.wraps(func)
  def network_fn(images):
    with slim.arg_scope(arg_scope):
      return func(images, num_classes, is_training=is_training)
  if hasattr(func, 'default_image_size'):
    # 每个模型的实现函数都会设定该模型的默认输入图像的分辨率（或大小）
    network_fn.default_image_size = func.default_image_size
  return network_fn
```

当我们要增加一个自定义模型时，首先需在 nets_factory 模块的 networks_map 和 arg_scopes_map 中添加一个键值对，然后在 nets 包中增加一个以模型名字命名的模块。下面以 AlexNet v2 模型为例，介绍 alexnet 模块包含的主要内容。该模块的主要实现代码如下：

```
def alexnet_v2(inputs,
               num_classes=1000,
               is_training=True,
               dropout_keep_prob=0.5,
               spatial_squeeze=True,
               scope='alexnet_v2'):
  with tf.variable_scope(scope, 'alexnet_v2', [inputs]) as sc:
    end_points_collection = sc.name + '_end_points'
    # tensorflow.contrib.slim.arg_scope(list_ops_or_scope, **kwargs)方法
    # (tensorflow.contrib.framework.python.ops.argscope(list_ops_or_scope,**kwargs)
    # 方法）的作用是：
    # 当 list_ops_or_scope 为 ops 列表时，
    # 将 kwargs 中指定的键值对作为 ops 列表中每个 Op 的输入参数。
    # 此处，调用 slim.arg_scope 方法，使得卷积（conv2d）、全连接（fully_connected）
    # 和最大池化（max_pool2d）三种算子的输出神经元组成的张量都保存在 end_points_collection 中
    with slim.arg_scope([slim.conv2d, slim.fully_connected, slim.max_pool2d],
                        outputs_collections=[end_points_collection]):
      net = slim.conv2d(inputs, 64, [11, 11], 4, padding='VALID',scope='conv1')
      net = slim.max_pool2d(net, [3, 3], 2, scope='pool1')
      net = slim.conv2d(net, 192, [5, 5], scope='conv2')
      net = slim.max_pool2d(net, [3, 3], 2, scope='pool2')
      net = slim.conv2d(net, 384, [3, 3], scope='conv3')
      net = slim.conv2d(net, 384, [3, 3], scope='conv4')
      net = slim.conv2d(net, 256, [3, 3], scope='conv5')
      net = slim.max_pool2d(net, [3, 3], 2, scope='pool5')
      # 利用 conv2d 实现全连接层
      with slim.arg_scope([slim.conv2d],
                          weights_initializer=trunc_normal(0.005),
```

```
                         biases_initializer=tf.constant_initializer(0.1)):
        net = slim.conv2d(net, 4096, [5, 5], padding='VALID', scope='fc6')
        net = slim.dropout(net, dropout_keep_prob, is_training=is_training, scope='dropout6')
        # 因为前一层已经是全连接层，即特征图的尺寸为 1×1，所以此处卷积核也设置为 1×1
        net = slim.conv2d(net, 4096, [1, 1], scope='fc7')
        net = slim.dropout(net, dropout_keep_prob, is_training=is_training, scope='dropout7')
        net = slim.conv2d(net, num_classes, [1, 1],
                         activation_fn=None,
                         normalizer_fn=None,
                         biases_initializer=tf.zeros_initializer,
                         scope='fc8')
      # 将 end_points_collection 转换为 end_points 字典。该字典中的 key 表示该模型中每一层的名字，
      # value 表示每一层的输出张量。
      end_points = slim.utils.convert_collection_to_dict(end_points_collection)
      if spatial_squeeze:
        # 如果 spatial_squeeze 为 True，则对最后一个全连接层的输出张量的形状进行转换，
        # 这样做的好处是能够去除一些不必要的维度，方便计算。该张量原来的形状为 [N,H,W,C]，
        # N 表示当前训练的批大小（batch_size），H 和 W 表示特征图的高和宽，C 表示特征图的通道数
        # （即分类问题中的类别数）。因为该张量是全连接层的输出，所以 H 和 W 都为 1。
        # 通过 tf.squeeze 方法将 H 和 W 所在的维度去掉，该张量的形状变为 [N,C]
        net = tf.squeeze(net, [1, 2], name='fc8/squeezed')
        end_points[sc.name + '/fc8'] = net
      return net, end_points
# AlexNet v2 模型的输入图像分辨率默认为 224×224
alexnet_v2.default_image_size = 224
```

通过 `is_training` 变量，我们可以设置该模型是用于训练还是推理计算。`alexnet_v2` 方法调用 `tensorflow.contrib.layers.python.layers` 中的二维卷积计算 API——`conv2d`、`max_pool2d`、`dropout` 分别实现卷积层、最大池化层和 Dropout 层的计算。以第一层卷积计算为例，`conv2d` 的第一个参数 `inputs` 表示该层的输入神经元，即输入图像；第二个参数 64 表示第一层卷积计算的输出为 64 通道；第三个参数 `[11,11]` 表示卷积核大小为 11×11；第四个参数 4 表示卷积核在输入图像上滑动的间距为 4；第五个参数表示补零的方式为 VALID 方式（详细解释见 9.1 节），如果不设置这个参数，补零方式将自动采用 `alexnet_v2_arg_scope` 中设定的默认参数值。

除了模型定义本身外，`alexnet` 模块还包括 `alexnet_v2_arg_scope` 方法，它用于提供模型的参数作用域 `arg_sc`。这样做的好处在于可以避免为每个算子（`conv2d` 等）显式指定很多参数。例如，在 AlexNet v2 模型中，所有激活层都采用 ReLU 激活函数，`alexnet_v2_arg_scope` 方法已经指定了激活函数为 `tf.nn.relu`，这使得 `alexnet_v2` 不需要在调用每个算子时再指定激活函数。

`alexnet_v2_arg_scope` 方法的代码及其解释如下：

```
def alexnet_v2_arg_scope(weight_decay=0.0005):
  # 在该参数作用域下的 conv2d 和 fully_connected 的计算中，激活函数默认采用
  # tf.nn.relu，biases 参数默认采用 0.1 恒定值作为初始化值，
  # weights 的正则化默认采用 L2 范式（可以防止过拟合）。用户可显式指定其他方式
  with slim.arg_scope([slim.conv2d, slim.fully_connected],
                      activation_fn=tf.nn.relu,
                      biases_initializer=tf.constant_initializer(0.1),
```

```
                  weights_regularizer=slim.l2_regularizer(weight_decay)):
# 在该参数作用域下的 conv2d 计算中，补零默认采用 SAME 方式，用户可显式指定其他方式
with slim.arg_scope([slim.conv2d], padding='SAME'):
  # 在该参数作用域下的 max_pool2d 中，补零默认采用 VALID 方式，用户可显式指定其他方式
  with slim.arg_scope([slim.max_pool2d], padding='VALID') as arg_sc:
    return arg_sc
```

9.2.6　TensorFlow-Slim最佳实践

TensorFlow-Slim 提供两个可作为程序执行入口的 Python 文件——train_image_classifier.py 和 eval_image_classifier.py，它们分别用于训练和评估深度学习模型。下面将简单介绍如何使用 TensorFlow-Slim 做 CNN 模型训练和推理。

在 `train_image_classifier` 模块中，程序首先使用 `tf.app.flags` 定义了一系列参数（其中 `FLAGS` 成员包含所有命令行参数），包括系统参数（比如模型克隆的个数、参数服务器个数、是否用 CPU 训练、当前计算节点的进程 ID 等），数据 IO 配置参数（比如 `reader` 的个数、预处理线程的个数），训练过程的配置参数（比如训练日志的保存间隔、summary 节点的数据保存间隔等），模型优化器相关的参数（比如参数权重衰减、优化器类型、动量值等），学习速率相关的参数（比如学习速率衰减模式、学习速率大小等），数据集相关的参数（比如数据集名、数据集类型、标签偏移、批大小、模型名等），以及微调模型时的参数（比如预训练模型的路径、微调模型时需要训练的参数等）。

随后，程序开始训练模型。训练过程的主要步骤包括定义部署配置、选择数据集、选择模型、选择预处理函数、建立数据提供器、建立模型、设置优化器和同步优化器（如需分布式训练）、获得单步训练操作并启动训练。各个步骤的具体代码及解释如下。

(1) 定义部署配置。根据自定义的 `flags` 参数生成一个 `DeploymentConfig` 对象。因为当前 `train_image_classifier` 模块仅提供单机单 GPU 或单机多 GPU 训练模式，暂时不支持分布式训练模式，所以 `FLAGS.task` 默认为 0，`FLAGS.worker_replicas` 和 `FLAGS.num_ps_tasks` 默认分别为 1 和 0。相关代码如下：

```
deploy_config = model_deploy.DeploymentConfig(
    num_clones=FLAGS.num_clones, # Clone 对象的个数
    clone_on_cpu=FLAGS.clone_on_cpu, # 布尔类型变量，表示是否将 Clone 对象部署在 CPU 上
    replica_id=FLAGS.task, # worker 或 PS 进程的 ID
    num_replicas=FLAGS.worker_replicas, # worker 任务数（详见 5.2 节）
    num_ps_tasks=FLAGS.num_ps_tasks) # PS 任务数
```

然后，新建一个同步标记 `global_step` 来对训练过程中的迭代次数进行计数：

```
with tf.device(deploy_config.variables_device()):
  global_step = slim.create_global_step()
```

(2) 选择训练数据集、模型函数和相应的预处理函数。在数据集工厂（`dataset_factory`）、网络工厂（`network_factory`）和数据预处理工厂（`preprocessing_factory`）中选择所需要的数据集、模型函数和数据预处理函数，供后续步骤使用：

```
# 根据 FLAGS 指定的数据集名字 dataset_name（如 imagenet）、
# 数据集被分割后的子数据集名称 dataset_split_name（如 train）
# 和数据集所在的绝对路径 dataset_dir，从 dataset_factory 中获得数据集对象 dataset
dataset = dataset_factory.get_dataset(
    FLAGS.dataset_name, FLAGS.dataset_split_name, FLAGS.dataset_dir)
# 根据 FLAGS 指定的模型名称 model_name（如 alexnet_v2）、
# 分类类别数 num_classes 和权值衰减 weight_decay（即 L2 正则项前面的系数），
# 从 nets_factory 中获得模型函数对象 network_fn
network_fn = nets_factory.get_network_fn(
    FLAGS.model_name,
    num_classes=(dataset.num_classes - FLAGS.labels_offset),
    weight_decay=FLAGS.weight_decay,
    is_training=True)
# 指定预处理函数名
preprocessing_name = FLAGS.preprocessing_name or FLAGS.model_name
# 根据预处理函数名，从 preprocessing_factory 中获得图像预处理函数对象
# image_preprocessing_fn。
image_preprocessing_fn = preprocessing_factory.get_preprocessing(preprocessing_name,
    is_training=True)
```

(3) 建立数据提供器。根据上一步获得的 `dataset` 对象，创建 `DatasetDataProvider` 对象，以得到训练所需要的输入图像和标签张量。为了加速数据集的读取性能，可以采用多线程方式读取数据。`DatasetDataProvider` 对象调用多个 `Reader` 对象的 `reader` 方法读取所需的数据：

```
with tf.device(deploy_config.inputs_device()):
  # FLAGS.num_readers 指定了同时读取数据集的线程数（默认为 4），
  # 不同线程读取的数据入队到 common_queue 中。此处默认设定 common_queue
  # 的最大容量为训练批大小（batch_size）的 20 倍。common_queue_min
  # 表示 common_queue 队列中最少保留的数据量，默认设定为训练批大小的 10 倍
  provider = slim.dataset_data_provider.DatasetDataProvider(
      dataset,
      num_readers=FLAGS.num_readers,
      common_queue_capacity=20 * FLAGS.batch_size,
      common_queue_min=10 * FLAGS.batch_size)
  # 如 9.2.1 节所述，可以根据 key 值 image 和 label 从 provider 对象中获得训练数据及其标签张量
  [image, label] = provider.get(['image', 'label'])
  # 因为在 VGG 或 ResNet 模型中，背景没有被当作分类数据集中的一个类别，
  # 所以当训练这两类模型时，labels_offset 要被设置为 1
  label -= FLAGS.labels_offset
  # 设定训练时输入图像的分辨率
  train_image_size = FLAGS.train_image_size or network_fn.default_image_size
```

当输入图像经被选定的预处理函数处理后，调用 `tf.train.batch` 方法得到当前迭代所用到的训练数据（批大小为 `FLAGS` 指定的 `batch_size`），并用 One-Hot 编码方式对 `labels` 进行编码。然后，调用 `slim.prefecth_queue.prefetch_queue` 方法得到 `batch_queue`，用于存放准备好的数据。具体代码如下：

```
# 训练数据经过图像预处理函数处理
image = image_preprocessing_fn(image, train_image_size, train_image_size)
# 通过 FLAGS.num_preprocessing_threads 指定的线程数并行读取，
# 得到当前迭代用到的训练数据 images 和 labels 张量
images, labels = tf.train.batch(
    [image, label],
```

```
        batch_size=FLAGS.batch_size,
        num_threads=FLAGS.num_preprocessing_threads,
        capacity=5 * FLAGS.batch_size)
    labels = slim.one_hot_encoding(
        labels, dataset.num_classes - FLAGS.labels_offset)
    # 调用 prefetch_queue 方法，启动一个 QueueRunner 对象用于保存预先准备好、
    # 即将被训练的数据。准备好的数据放在缓冲区队列 batch_queue 中
    batch_queue = slim.prefetch_queue.prefetch_queue(
        [images, labels], capacity=2 * deploy_config.num_clones)
```

(4) 定义 `clone_fn` 方法。根据之前选定的某个具体的模型函数（如 AlexNet v2 等）和相应的批数据（图像 `images` 和标签 `labels`）得到模型各层的输出：

```
def clone_fn(batch_queue):
  # 从 batch_queue 中得到本次迭代所需要的训练数据————images 和 labels
  images, labels = batch_queue.dequeue()
  # 调用 network_fn，得到 CNN 模型最后一层的输出张量 logits，
  # 以及由 CNN 模型中每层的输出张量所组成的集合 end_points
  logits, end_points = network_fn(images)
  # 在某些 CNN 模型（如 Inception V3）中，为了减少梯度消失现象，
  # 模型中间某一个或多个层的输出被用于辅助分类。这些层的输出张量为 AuxLogits
  if 'AuxLogits' in end_points:
    # 将辅助分类层的损失函数值也计算在模型整体的损失值中。
    # weight 参数表示辅助分类层对应的损失值在计入总损失值时被乘的折扣系数
    slim.losses.softmax_cross_entropy(
        end_points['AuxLogits'], labels,
        label_smoothing=FLAGS.label_smoothing, weight=0.4, scope='aux_loss')
  # 计算最后分类层所对应的损失值
  slim.losses.softmax_cross_entropy(
      logits, labels, label_smoothing=FLAGS.label_smoothing, weight=1.0)
  # 返回模型每层的输出张量所组成的集合
  return end_points
```

(5) 建立训练模型。根据已定义的 `DeploymentConfig` 对象、模型函数和训练数据的预取队列，建立相应个数的模型 `Clone` 对象组成的列表——`clones`：

```
clones = model_deploy.create_clones(deploy_config, clone_fn, [batch_queue])
```

根据第一个 `Clone` 对象的作用域得到一系列更新操作，这些更新操作与每次迭代时模型参数的更新有关：

```
first_clone_scope = deploy_config.clone_scope(0)
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS, first_clone_scope)
```

如果 `FLAGS` 中指定了 `moving_average_decay` 的值，那么需要对该模型中的模型参数做滑动平均操作，以提升模型的健壮性。假设 t 时刻模型参数 w_i 的值为 $w_i(t)$，则此时与 w_i 关联的滑动平均变量被定义为 $m_i(t)=d\cdot m_i(t-1)+(1-d)\cdot w_i(t)$。其中，$d$ 表示衰减率，它是一个可被调节的超参，由 `FLAGS.moving_average_decay` 指定。当 d 越大时，模型参数的更新越慢。相关代码如下：

```
if FLAGS.moving_average_decay:
  # 如果衰减率 FLAGS.moving_average_decay 的值被指定，
  # 则 moving_average_variables 表示具有滑动平均特性的模型参数变量，
  # variable_averages 表示相应的滑动平均变量
```

```
        moving_average_variables = slim.get_model_variables()
        variable_averages = tf.train.ExponentialMovingAverage(
                            FLAGS.moving_average_decay, global_step)
    else:
      moving_average_variables, variable_averages = None, None
```

(6) 定义模型的优化器。首先根据 FLAGS 所指定的梯度衰减策略，创建相应的学习速率张量 learning_rate。然后根据 learning_rate 和 FLAGS 所指定的优化器类型创建优化器 optimizer：

```
with tf.device(deploy_config.optimizer_device()):
  # 当前，在学习速率的调整方面，支持 exponential、fixed、polynomial 这三种策略
  learning_rate = _configure_learning_rate(dataset.num_samples, global_step)
  # 根据 FLAGS 所指定的优化器类型创建相应的优化器 optimizer。
  # 当前支持 adadelta、adagrad、adam、ftrl、momentum、rmsprop 和 sgd 这七种优化器
  optimizer = _configure_optimizer(learning_rate)
  summaries.add(tf.scalar_summary('learning_rate', learning_rate,
                                  name='learning_rate'))
```

对于多个 worker 并行训练的情况，我们可以定义一个同步优化器，利用 QueueRunner 来保存每个 worker 计算得到的梯度，并对模型梯度求平均。代码如下：

```
if FLAGS.sync_replicas:
  # 如 5.2 节所述，进行分布式计算时，需要定义同步优化器。
  # 当前开源的 train_image_classifier.py 对分布式支持还不完善，
  # 此处代码需要配合 tf.train.ClusterSpec、tf.train.Server 等接口一起使用，才能实现分布式训练
    optimizer = tf.train.SyncReplicasOptimizer(
        opt=optimizer,
        replicas_to_aggregate=FLAGS.replicas_to_aggregate,
        variable_averages=variable_averages,
        variables_to_average=moving_average_variables,
        replica_id=tf.constant(FLAGS.task, tf.int32, shape=()),
        total_num_replicas=FLAGS.worker_replicas)
elif FLAGS.moving_average_decay:
  # 如果衰减率 FLAGS.moving_average_decay 的值被指定，则对模型参数更新采取滑动平均操作
  update_ops.append(variable_averages.apply(moving_average_variables))
```

(7) 获得单步训练操作并启动训练。根据之前定义的 optimizer、clones，以及模型参数，调用 model_deploy.optimize_clones 方法得到的多个模型克隆的损失值和梯度。进而通过调用 optimizer.apply_gradients 方法得到梯度更新操作,然后将这些梯度更新操作合并为 train_tensor：

```
# 此段代码类似于 9.2.3 节介绍的 deploy 方法的部分代码（当 optimizer 非 None 时）
variables_to_train = _get_variables_to_train()
total_loss, clones_gradients = model_deploy.optimize_clones(clones, optimizer,
                                                            var_list=variables_to_train)
summaries.add(tf.scalar_summary('total_loss', total_loss, name='total_loss'))
grad_updates = optimizer.apply_gradients(clones_gradients, global_step=global_step)
update_ops.append(grad_updates)
update_op = tf.group(*update_ops)
train_tensor = control_flow_ops.with_dependencies([update_op], total_loss, name='train_op')
```

最后，调用 slim.learning.train 方法启动训练：

```
slim.learning.train(
    train_tensor, # 单步迭代的训练操作
    logdir=FLAGS.train_dir, # 训练过程中日志和模型检查点文件等存放的目录
    master=FLAGS.master, # master 的地址，在单机训练时没有用到
```

```
        is_chief=(FLAGS.task == 0),# 当前 worker 是否为 chief worker（在分布式训练场景中用到）
        init_fn=_get_init_fn(), # 模型初始化函数
        summary_op=summary_op, # summary 操作
        number_of_steps=FLAGS.max_number_of_steps, # 最大训练步数
        log_every_n_steps=FLAGS.log_every_n_steps, # 输出日志的间隔（以步数为单位）
        save_summaries_secs=FLAGS.save_summaries_secs, # 输出 summary 日志的间隔（以秒为单位）
        save_interval_secs=FLAGS.save_interval_secs, # 保存模型检查点文件的间隔（以秒为单位）
        sync_optimizer=optimizer if FLAGS.sync_replicas else None # 同步优化器（在单机训练时为 None）
```

在模型推理阶段，我们需要使用 eval_image_classifier 模块。该模块的处理流程与 train_image_classifier 类似，但因为它不需要训练，所以没有涉及优化器的定义等环节。用户可以在 eval_image_classifier 模块中指定 Accuracy 和 Recall，用来对模型的好坏作出评价。相关代码如下：

```
# 定义 Accuracy 和 Recall
names_to_values, names_to_updates = slim.metrics.aggregate_metric_map({
    'Accuracy': slim.metrics.streaming_accuracy(predictions, labels),
    'Recall@5': slim.metrics.streaming_recall_at_k(logits, labels, 5),
})
```

我们可以通过调用 slim.evaluation.evaluate_once 方法实现模型推理：

```
slim.evaluation.evaluate_once(
    master=FLAGS.master, # master 的地址
    checkpoint_path=checkpoint_path, # 模型检查点文件所存放的目录
    logdir=FLAGS.eval_dir, # 推理过程中输出日志所存放的目录
    num_evals=num_batches, # 推理过程中所计算的批的个数
    eval_op=names_to_updates.values(), # 推理态所需执行的操作
    variables_to_restore=variables_to_restore) # 在推理过程中所要加载参数值的变量
```

9.3　应用

9.1 节提到的模型都是分类模型，这些模型的输入是图像，输出是图像的类别。但事实上，除了可以应用于图像分类之外，CNN 还可以用于检测、分割等其他应用。例如，在很多实际应用中，我们不仅需要得到整个图像的分类结果，还要得到图像中每个物体的位置、大小和类别信息，这就需要物体检测模型。在一些机器人视觉导航系统中，我们需要对周围环境的图像做分割，即确定每个像素的标签，以实现场景理解。

9.3.1　物体检测

物体检测是图像和视频处理技术的一类重要应用，典型的应用场景之一是视频监控中的行人检测。在数据集方面，不同于图像分类数据集，物体检测数据集中训练集的标注比较复杂。标注信息包括图像中包含的每个物体的类别及其包围框（bounding box）的左上角和右下角的坐标，我们可以根据这些坐标计算出物体在图像中的位置和大小。通常，IoU（Intersection over Unit）用于表示预测的包围框和标注的包围框的相似度，它被定义为这两个包围框的交集面积与并集面积之比。当 IoU=1 时，预测的框与标注框重合。

在算法和模型方面，以行人检测为例，以前比较主流的做法是采用 HOG 算子作为特征提取

器，并以 SVM 等作为分类器。当前，因为经过大量数据训练的 CNN 模型的特征提取能力很强，所以主流的物体检测算法都是基于 CNN 模型构建的。常用的算法模型有以下几种。

1. R-CNN 模型

R-CNN（Regions with CNN）首次将 CNN 模型应用于物体检测领域，并在 Pascal VOC 公开数据集上验证了其有效性。如图 9-17 所示，R-CNN 模型首先通过 SS（Selective Search）方法提取图像中所有可能存在的物体候选框（约 2000 个），然后将每个物体所占像素面积归一化为某一 CNN 模型的输入尺寸（如 227×227），并使其经过 CNN 做特征提取，将提取的特征向量输入到 SVM 分类器中做分类，以得到每个物体候选框所对应的物体类别。为了得到更精确的物体位置信息，R-CNN 模型包含位置回归模型，其输入为 SS 方法提取的物体中心坐标 P_x、P_y，物体的宽度 P_w 和高度 P_h，以及相应物体包围框的标注信息；输出为每个物体中心坐标的预测值 G_x、G_y，以及物体的宽度、高度的预测值 G_w 和 G_h。

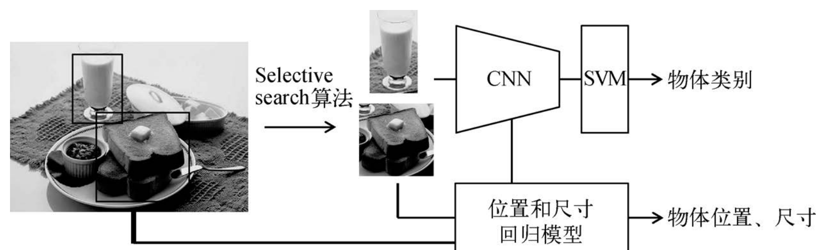


图 9-17　R-CNN 模型的基本框架（另见彩插）

R-CNN 模型利用了迁移学习的思想。由于物体检测领域内的开源标注数据集相对较少，R-CNN 中用于特征提取的大部分 CNN 参数来自已经训练好的 CNN 分类网络。在 Pascal VOC 数据集上，我们只需基于这些参数再做微调。

9

2. SPPNet 模型

R-CNN 模型的 SS 方法在每个输入图像上提取出约 2000 个候选框，候选框之间往往存在大量的重叠，每个重叠区域都要重复地进行 CNN 特征提取。这样一来，R-CNN 模型就会包含大量的重复计算。为了解决这个问题，有人提出了 SPPNet。该模型引入了 SPP（Spatial Pyramid Pooling）层。SPP 是基于 Bag-of-words 的图像分类方法（该方法依据图像中以 SIFT 特征为核心建立的“视觉词汇”分布图对图像分类）的一个延伸，它的核心思想是考虑图像中空间位置的重要性。如图 9-18 所示，该模型首先使用 CNN 网络中的卷积层对原始图像做特征提取，同时采用 SS 方法在原始图像中提取一系列候选框，并在 CNN 提取得到的特征图中找到原始图像中每个候选框所对应的特征子图。接下来，对每个特征子图进行 SPP 处理，即将特征子图按照固定规格的分块做池化。比如，将每个特征子图分别分割为 $M×N$ 块，然后将每一块池化为 SPP 层输出的特征图中的一个点。M 和 N 的值可以被改变以产生不同粒度（对应图像金字塔的不同层）的池化结果。不同粒度的池化结果组合起来形成 SPP 层的所有输出特征图。这些输出特征图进入后续的全连接层做进

一步特征提取后，产生固定长度的特征向量，便于最终的分类。

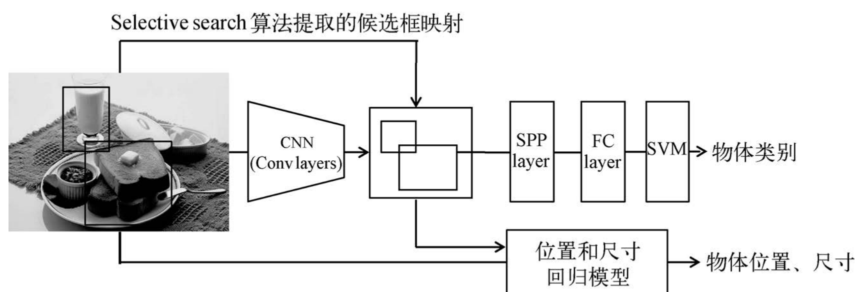


图 9-18　SPPNet 模型的基本框架（另见彩插）

SPP 层的引入使得 SS 方法产生的不同大小的候选框在进入全连接层之前具有相同的特征维度。SPPNet 避免了多个候选框间重复部分的卷积计算，在计算速度上比 R-CNN 模型有两个数量级的提升。与 R-CNN 模型相同的是：在用 SVM 对每个候选框做完分类后，SPPNet 模型对每个候选框的位置和大小做进一步的回归拟合，使得对物体的定位更准确。

3. Fast R-CNN 模型

R-CNN 模型和 SPPNet 模型在处理流程上仍然遵循着提取特征、微调网络、训练 SVM 分类器、拟合候选框位置和尺寸值这四个步骤，而不是端到端的训练过程。虽然 SPP 层是可微的，但是 SPPNet 也没有对 SPP 层之前的卷积层做微调更新。为了解决这些问题，Ross Girshickt 提出了 Fast R-CNN 模型，其基本框架如图 9-19 所示。与 SPPNet 不同，Fast R-CNN 模型采用 ROI（Region of Interest）池化层，将不同大小的候选框对应的特征图转换为固定大小（$H \times W$）的特征图。ROI 池化层可以看作 SPP 层的一个特例，即仅有一个金字塔层的情况（HW 的取值仅有一种）。在 ROI 池化层和全连接层之后，该模型有两个输出层，一个是用于物体分类的 Softmax 层，另一个是用于输出物体位置和尺寸的回归层。

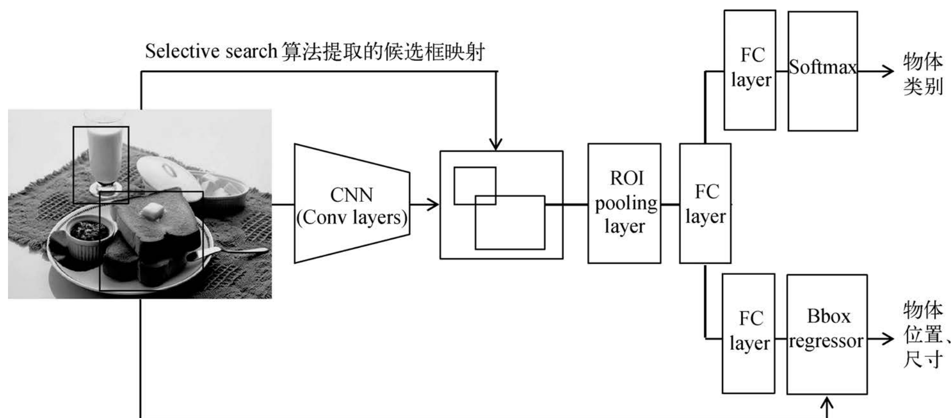


图 9-19　Fast R-CNN 模型的基本框架（另见彩插）

4. Faster R-CNN 模型

在 R-CNN、SPPNet 和 Fast R-CNN 中，目标物体的候选框提取算法均为 SS 方法。在该方法中，候选框是图像经过分割之后的超像素区域不断相互融合所形成的。实际应用时，SS 方法非常耗时。为了进一步提升目标检测算法的实时性，Faster R-CNN 模型首次引入 RPN（Region Proposal Network）来取代 SS 方法以产生候选框。如图 9-20 所示，在 RPN 的输入特征图中，我们假设以每个像素点为中心都存在不同面积大小和长宽比的参考框（anchor）。默认情况下，参考框长宽组合有 $(128, 128)$、$(\frac{128}{\sqrt{2}}, 128\sqrt{2})$、$(128\sqrt{2}, \frac{128}{\sqrt{2}})$、$(256, 256)$、$(\frac{256}{\sqrt{2}}, 256\sqrt{2})$、$(256\sqrt{2}, \frac{256}{\sqrt{2}})$、$(512, 512)$、$(\frac{512}{\sqrt{2}}, 512\sqrt{2})$、$(512\sqrt{2}, \frac{512}{\sqrt{2}})$ 这九种（如果值为小数，则取整）。当我们要对预测的候选框中心位置和大小信息做参数回归时，这些 anchor 起到了中间桥梁的作用。在 RPN 中，特征图先经过一层卷积（带激活函数）计算，再分别经过两个并行的卷积层得到输出张量。其中一个卷积层的输出张量表示每个 anchor 内的物体是目标（前景）还是背景，另外一个的输出张量表示候选框的中心位置和大小信息。

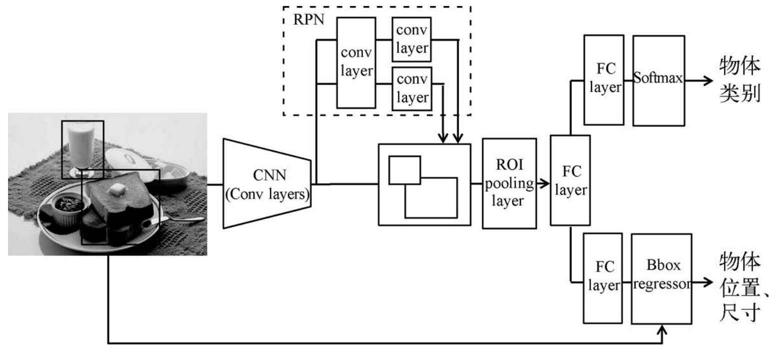


图 9-20　Faster R-CNN 模型的基本框架

Faster R-CNN 中的 RPN 损失函数定义为：$L(p_i,t_i)=\frac{1}{N_{\text{cls}}}\sum_i L_{\text{cls}}(p_i,p_i^*)+\lambda\frac{1}{N_{\text{reg}}}\sum_i p_i^* L_{\text{reg}}(t_i,t_i^*)$。其中，$N_{\text{cls}}$ 表示每次迭代所用到的批大小；N_{reg} 表示 RPN 输入特征图中 anchor 的总数；p_i 表示第 i 个 anchor 中包含物体的概率（$0 \leqslant i < N_{\text{cls}}$）；$p_i^*$ 是标注信息。当第 i 个 anchor 与任意一个标注的包围框的 IoU 值大于 0.7（一个既定阈值，可根据实际数据集进行调节），或者与某个标注的包围框有最大的 IoU 值时，$p_i^*=1$，否则 $p_i^*=0$。$t_i=[t_x, t_y, t_w, t_h]$ 和 $t_i^*=[t_x^*, t_y^*, t_w^*, t_h^*]$ 分别表示预测的候选框和标注的包围框的位置、尺寸信息（即中心坐标值 x、y 和宽度 w、高度 h）在经过归一化之后的值，具体的计算公式为：$t_x=(x-x_a)/w_a$，$t_y=(y-y_a)/h_a$，$t_w=\log(w/w_a)$，$t_h=\log(h/h_a)$，$t_x^*=(x^*-x_a)/w_a$，$t_y^*=(y^*-y_a)/h_a$，$t_w^*=\log(w^*/w_a)$，$t_h^*=\log(h^*/h_a)$。其中，变量的下角标为 a，表示该变量用于表征第 i 个 anchor 的相关参数；变量的上角标为*，表示该变量用于表征标注的

包围框的相关参数；没有上角标或下角标的变量用于表征 RPN 预测的候选框的相关参数。

最后，原始图像经 CNN 处理后的特征图连同 RPN 产生的多个候选框输入到 Fast R-CNN 中，就可以实现最终物体的分类和物体位置、大小的检测。

5. YOLO 模型

与上述所有模型单独采用某些算法或网络提取目标物体候选框的做法不同，YOLO（You Only Look Once）模型将原始输入图像直接均匀划分为 $S\times S$ 大小的网格（一般取 S=7），然后将其输入到带有卷积层、全连接层的 CNN 网络中。输入图像中，每个网格不仅负责预测 B 个候选框的 5 维信息（一般取 B=2；5 维信息中 4 维是候选框的位置 P_x, P_y 和尺寸信息 P_w, P_h，剩下一维表示该候选框内是否包含有目标物体的概率值 $\text{Prob(Object)} \times \text{IoU}^{\text{truth}}_{\text{prediction}}$），还负责预测 C 个物体的类别信息 $\text{Prob}(\text{Class}_i|\text{Object})$（对于 Pascal VOC 数据集，$C = 20$）。因此 YOLO 模型的输出为 $S \times S \times (5B + C)$个预测值。YOLO 模型的基本框架如图 9-21 所示。

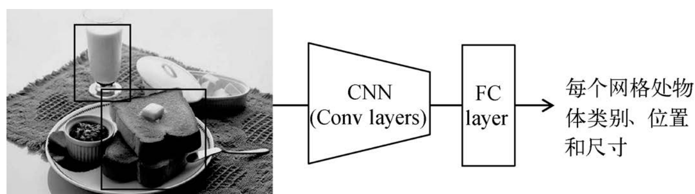


图 9-21　YOLO 模型的基本框架

YOLO 模型的优点在于其速度极快，比 Faster R-CNN 至少高一个数量级，特别适合于端侧（如手机端）的目标检测。但是由于输入图像事先被划分比较粗粒度的 $S\times S$ 个窗格，所以 YOLO 模型不太适合做小物体检测。另外，该模型在每个窗格处仅预测 2 个候选框，所以对于一些多个物体重叠的场景可能会失效。

6. SSD 模型

类似于 YOLO 模型，SSD 模型直接采用一个 CNN 网络同时完成对每个候选框中物体类别的预测及目标物体位置和尺寸的回归预测。不同的是，SSD 模型考虑了多尺度的特征图对最终预测的影响。如图 9-22 所示，在经过基础的几层卷积之后，SSD 模型采用额外的多个卷积层。每层卷积的输出通道数为 $D\times(C+4)$，其中 D 表示在原始图像中每个窗格内默认候选框的个数（假设原始图像被分为 8×8 或者 4×4 个窗格）。

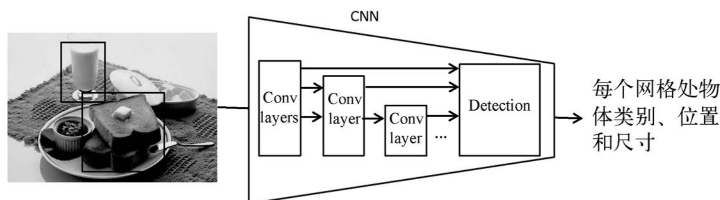


图 9-22　SSD 模型的基本框架

SSD 模型的优点在于比 YOLO 模型更快，在很多公开数据集上更准。然而，由于事先在原始输入图像中采用网格划分的方式提取候选框，SSD 和 YOLO 一样不太适用于检测小物体。

在物体检测领域中还有其他模型，比如早期的 OverFeat 和后来的 FPN（Feature Pyramid Network）、PVANet 等，这些模型在小物体检测、模型小型化、检测的实时性方面做了一定改进。

9.3.2 图像分割

图像分割是指给图像中每个像素点都提供分类标签的一种图像处理技术。常用的图像分割方法有语义分割法、实例分割法等。语义分割法是指为输入图像中每个像素点标记相应类别的图像分割方法；实例分割法是指为输入图像分割出其中每个目标物体实例的图像分割方法。对于输入图像中属于同一类别的多个目标物体实例，实例分割法需要将它们区分开，而语义分割则不需要区分。下面介绍三个典型模型，其中 FCN 和 SegNet 属于图像语义分割模型，Mask R-CNN 属于实例分割模型。

1. FCN 模型

含有全连接层的 CNN 模型对输入图像的尺寸有严格要求。一旦输入图像尺寸有变化，全连接层的输入神经元个数就会变化，这会造成全连接层输入尺寸不匹配的问题。如果将全连接层替换为卷积层，即采用全卷积网络，则输入图像的大小可以为任意尺寸。FCN（Fully Convolutional Network）是最早将深度学习用于图像语义分割的经典模型，该模型没有全连接层，输入图像的尺寸是可变的。如图 9-23 所示，FCN 模型将多个卷积层输出的结果再反卷积，就得到了原始图像的语义分割结果。FCN 模型将不同卷积层的输出相加，使得不同尺度下的图像信息可以融合。为保持不同层输出的特征图尺寸统一，需要适当地加以反卷积操作。在具体的模型训练过程中，我们可以采用分阶段训练方法，即不同尺度对应的反卷积层的参数依次被训练，这样可以降低模型训练的难度。

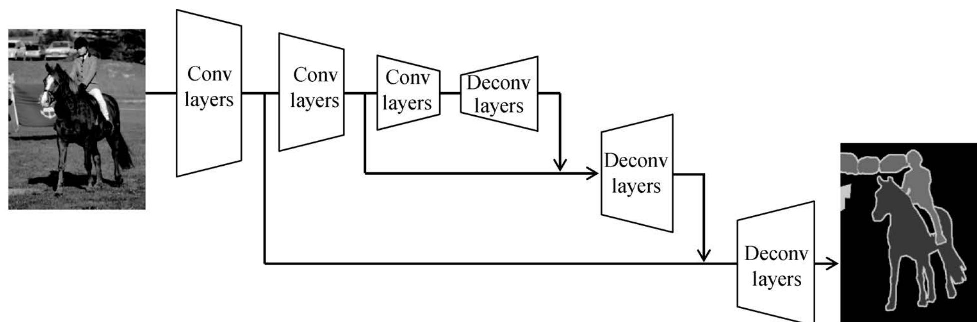


图 9-23 基于 FCN 的语义分割模型的基本框架（另见彩插）

2. SegNet 模型

SegNet 模型的整体架构类似于 FCN 模型，即基于编码器-解码器的架构，其中卷积层为编码

器，上采样和反卷积层为解码器。以 FCN 模型架构为基础，SegNet 模型做了进一步优化。SegNet 不同于 FCN 模型的细节在于：

- 解码器在上采样时，利用了编码器中池化时输入像素点的位置信息，这使得解码器中待分割物体的边界更清晰；
- 采用 BN 层加速训练；
- 在解码器中，卷积时没有偏置参数，没有 ReLU 非线性激活层；
- 在编码器和解码器中采用 7×7 的大卷积核，使得更上层的输出所用到的信息更多。

3. Mask R-CNN 模型

Mask R-CNN 模型是对 Faster R-CNN 模型的拓展，用于图像中的实例分割，该模型的整体架构类似于图 9-20。我们已在图 9-20 中看到，Faster R-CNN 的输出有两个分支，其中一个是预测每个候选框类别的 class 分支，另一个是预测每个候选框尺寸和位置的 bbox 分支。Mask R-CNN 在 Faster R-CNN 的基础上又添加了一个 mask 分支，它采用 FCN 对每个候选框内的每个像素做标签预测，即实例分割。Mask 分支首先会根据 Faster R-CNN 中的 class 分支确定该候选框内的物体类别，然后再判断该候选框内的每个像素是否属于该类别（本质上是像素级的二分类问题）。

除了物体检测和图像分割外，CNN 还可以用于图像处理领域的其他任务，如关键点检测、图像标题生成、目标跟踪等。为了更好地应用于不同任务，CNN 模型需要和其他图像处理算法相结合。在这些过程中，新的一些层和网络也被发明并引入到 CNN 模型中，比如物体检测中的 ROI 池化层和 RPN 网络。

9.4　小结

CNN 模型是深度学习中非常重要的一类模型，它适用于非结构化数据的特征提取。CNN 模型包含卷积层、全连接层、Dropout 层、池化层、BN 层等多种类型的层，常用的 CNN 模型包括 AlexNet、Inception、VGG、ResNet 等。这些模型中，Inception module 和 ResNet module 等模块的设计蕴含了发明者丰富的经验知识，考虑了精度和速度之间的平衡，值得我们在今后的模型设计中参考。TensorFlow-Slim 是基于 TensorFlow 原生 API 封装的一套 Python 库，它提供了数据集处理、模型定义、模型训练和推理在内的诸多功能，方便算法开发者快速实现各种神经网络。除了图像分类外，CNN 模型还可以用于其他计算机视觉任务，如物体检测、图像分割等。理解 CNN 模型的经典设计，有助于针对特定的问题开发专用的模型变种。

第 10 章

GAN 模型

10

机器学习和深度学习领域不仅有判别模型，而且有生成模型。判别模型与生成模型的区别在于：判别模型的目标是得到具有不同标签的数据之间的边界，使得机器能够找到不同数据分布之间的不同；而生成模型的目标是得到每个标签所对应的数据分布，使得机器可以理解如何得到或修改数据的分布。判别模型在分类、回归等问题上往往很实用，而生成模型更为复杂，它期望为机器赋予近乎人类的理解力和创造力。上一章介绍的 CNN 模型属于判别模型，本章将介绍一种生成模型——GAN（Generative Adversarial Network，生成对抗网络）。该模型可以利用深度学习强大的非线性拟合能力模拟真实的数据分布，从而生成绘画、照片、音乐等类型的逼真数据。

10.1　原理、特点及应用

我们首先回顾一下机器学习中的两种模型：判别模型（Discriminative Model）和生成模型（Generative Model）。在机器学习问题中，假设训练数据的特征集为 $\boldsymbol{X} \in \mathbf{R}^{m \times n}$，对应的标签集为 $\boldsymbol{y} \in \mathbf{R}^{m}$，特征维度为 n，训练数据样本个数为 m，模型参数为 $\boldsymbol{\theta}$。对条件概率 $P(\boldsymbol{y}|\boldsymbol{X};\boldsymbol{\theta})$ 直接建模而得到的模型就是判别模型。SVM（Support Vector Machine）、LR（Logistic Regression）等都属于判别模型。以分类问题为例，判别模型不关注每个类别所对应的数据分布形式，而直接在训练数据中寻找最优的分类面。与判别模型不同，生成模型是指对训练数据特征和标签的联合概率 $P(\boldsymbol{X},\boldsymbol{y};\boldsymbol{\theta})$ 建模而得到的模型。通过生成模型，可以知道每个类别所对应的数据分布情况。生成模型是无监督学习和半监督学习中的重要方法。生成模型有助于对更复杂现象建模，促进更高级人工智能的探索。

常用的生成模型主要包括两大类：基于显式概率密度函数的生成模型和基于隐式概率密度函数的生成模型。前者要求我们首先建立所需拟合的数据分布的概率密度函数，然后用有限的样本不断训练该模型。假设该模型 G 的目标优化函数为 $E_{\boldsymbol{x}\sim p_{\text{data}}}\log G(\boldsymbol{x}|\boldsymbol{\theta})$，其最优参数为 $\boldsymbol{\theta}^{*}=\operatorname{argmax}_{\boldsymbol{\theta}} E_{\boldsymbol{x}\sim p_{\text{data}}}\log G(\boldsymbol{x}|\boldsymbol{\theta})$。当该目标优化函数可解时，我们可以直接用优化算法求解该模型的最优参数。常见的此类模型有 FVBN（Fully Visible Belief Network）等。当它不可解时，我们需要用近似的方法得到该模型的最优参数。变分近似（Variational Approximations）和马尔可夫蒙特卡洛（Markov Chain Monte Carlo，简称 MCMC）算法是常用的两种近似算法。在变分近似法中，我们首先得到原始目标优化函数的一个可解的下限函数，然后通过求解该下限函数的最大值间接得到原始目标

优化函数的最大值，其代表模型为 VAE（Variational AutoEncoder）。然而，下限函数带来的近似有可能会使得生成模型所拟合的数据分布与真实的数据分布间存在偏差。在马尔可夫蒙特卡洛算法中，我们使用马尔可夫链不断产生采样以获得新数据，并使得经过足够多的迭代后采样得到的数据分布能够逼近真实的数据分布，其代表模型为玻尔兹曼机。MCMC 方法的计算复杂度通常比较高，并随着数据维度的增加而增加。不同于基于显式概率密度函数的生成模型，基于隐式概率密度函数的生成模型可以通过直接采样的方式训练模型参数，而不需要提前对真实数据的分布建立模型。与围绕前者的研究成果相比，业界对这类模型的研究还不是很深入，因此这类模型并不多见。在这类模型中，一般常用的采样方式基于马尔可夫链实现，但其计算复杂度会随训练数据维度的增加而增加。GAN 是一种全新的基于隐式概率密度函数的生成模型。下面分析 GAN 模型的原理和特点，并介绍其应用。

10.1.1　原理

为了解决已有生成模型的上述缺点，加拿大蒙特利尔大学 Yoshua Bengio 教授门下的 Ian Goodfellow 等人于 2014 年提出了 GAN 模型。如图 10-1 所示，该模型主要由两个子模型——生成器 G 和判别器 D 组成。G 的输入为随机噪声向量，输出为与真实数据维度相同的数据。D 的输入为真实数据或者 G 生成的数据，输出为其对输入数据的分类（用来判断输入数据是否为真实数据）。G 的目标是使其生成的数据与真实数据无限接近，而 D 的目标是尽可能分辨出输入数据的真伪，即将 G 生成的数据和真实数据区分开。因此，G 和 D 的优化目标相互对立，GAN 模型的优化过程就是 G 和 D 相互对抗的过程，GAN 的名字也由此而来。

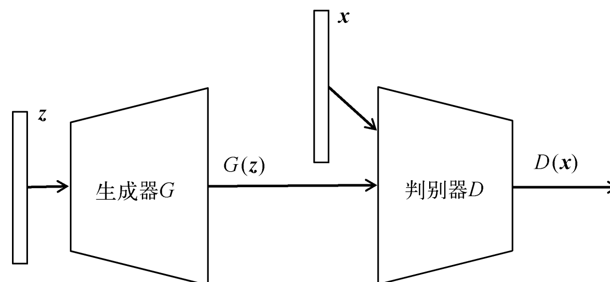


图 10-1　GAN 的原理框图

根据上述对抗过程，GAN 模型的目标优化问题可以表示为：

$$\min_G \max_D V(D, G)= E_{x\sim p_{\text{data}}}[\log D(\boldsymbol{x})]+E_{z\sim p_z(z)}[\log(1-D(G(\boldsymbol{z})))]$$

其中，D 输出值的区间为 [0,1]。D 的输入数据越接近真实数据，其输出值越接近 1。在实际中，我们采用多层卷积神经网络或者深度学习模型拟合生成器 G 和判别器 D，通过迭代的方式不断优化目标函数 $V(D, G)$。为了使 D 尽可能地保持在最优解状态，并且使 G 的参数逐步变化，我们在 GAN 模型优化过程的每一次迭代中先训练 k 次判别器 D，再训练一次生成器 G。假设每次批数据大小为 m，总的迭代次数为 N，那么训练 GAN 模型的具体步骤如下。

(1) 训练判别器。从噪声输入向量 $\boldsymbol{z}$ 的分布 $p_z(\boldsymbol{z})$中随机采样 m 个样本组成一组用于输入到生成器的批训练数据 $\{\boldsymbol{z}^{(1)},\cdots,\boldsymbol{z}^{(m)}\}$，从真实数据分布 $p_{\text{data}}(\boldsymbol{x})$ 中随机采样 m 个样本组成一组用于输入到判别器的批训练数据 $\{\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)}\}$。计算判别器的损失函数在当前的批训练数据上的平均值 $L_D=\frac{1}{m}\sum_{i=1}^{m}\left[\log D(\boldsymbol{x}^{(i)})+\log(1-D(G(\boldsymbol{z}^{(i)})))\right]$，然后求解 L_D 对判别器参数 $\boldsymbol{\theta}_d$ 的导数，用梯度上升法更新 $\boldsymbol{\theta}_d$。

(2) 重复执行步骤 (1)，共计执行 k 次。

(3) 训练生成器。从 $p_z(\boldsymbol{z})$ 中随机采样得到一组批训练数据 $\{\boldsymbol{z}^{(1)},\cdots,\boldsymbol{z}^{(m)}\}$。计算生成器的损失函数在当前的批训练数据上的平均值 $L_G=\frac{1}{m}\sum_{i=1}^{m}\log(1-D(G(\boldsymbol{z}^{(i)})))$，然后求解 L_G 对生成器参数 $\boldsymbol{\theta}_g$ 的导数，用梯度下降法更新 $\boldsymbol{\theta}_g$。

(4) 重复执行步骤 (1) ~ 步骤 (3)，共计执行 N 次。

对于判别器而言，需要被最优化的目标函数可以被描述为：

$$\begin{aligned}V_D(D,G) &= E_{\boldsymbol{x}\sim p_{\text{data}}}[\log D(\boldsymbol{x})] + E_{z\sim p_z(z)}\log(1-D(G(\boldsymbol{z})))\\ &=\int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x})\log(D(\boldsymbol{x}))\mathrm{d}x+\int_{\boldsymbol{z}} p_z(\boldsymbol{z})\log(1-D(g(\boldsymbol{z})))\mathrm{d}z\\ &=\int_{\boldsymbol{x}}\left[p_{\text{data}}(\boldsymbol{x})\log(D(\boldsymbol{x}))+p_g(\boldsymbol{x})\log(1-D(\boldsymbol{x}))\right]\mathrm{d}x\end{aligned}$$

其中，$p_{\text{data}}(\boldsymbol{x})$ 和 $p_g(\boldsymbol{x})$ 分别用于表示真实数据和生成数据的概率分布。我们可以将 $D(\boldsymbol{x})$ 看作一个未知量，令 $\frac{\mathrm{d}V_D(D,G)}{\mathrm{d}(D(\boldsymbol{x}))}=0$，即 $p_{\text{data}}(\boldsymbol{x})\frac{1}{D^*(\boldsymbol{x})}+p_g(\boldsymbol{x})\frac{1}{1-D^*(\boldsymbol{x})}=0$，可得出 $D^*(\boldsymbol{x})=\frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x})+p_g(\boldsymbol{x})}$ 为 $D(\boldsymbol{x})$的最优解。当 $D(\boldsymbol{x}) = D^*(\boldsymbol{x})$时，GAN 模型的目标函数变为 $\min_G V(D,G^*) = E_{\boldsymbol{x}\sim p_{\text{data}}}\left[\log\frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x})+p_g(\boldsymbol{x})}\right]+E_{z\sim p_z(z)}\left[\log\frac{p_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x})+p_g(\boldsymbol{x})}\right]$。该目标函数可进一步被表示为 $\min_G V(D,G^*)=-\log(4)+KL\left(p_{\text{data}}\|\frac{p_{\text{data}}+p_g}{2}\right)+KL\left(p_g\|\frac{p_{\text{data}}+p_g}{2}\right)=-\log(4)+2\text{JSD}(p_{\text{data}}\|p_g)$。其中，$KL(x\|y)$ 和 $\text{JSD}(x\|y)$ 分别表示概率分布 x 和 y 之间的 KL 散度（或者相对熵）和 JS 散度。KL 散度和 JS 散度均可用于描述两个分布之间的差别，其值非负。当概率分布 x 和 y 相同时，KL 散度和 JS 散度为 0，此时 $V(D,G^*)$ 取得最小值。因此，当生成器产生的数据的概率分布与真实数据的概率分布相同时，其目标函数达到最小值，模型参数为最优参数。

10

10.1.2 特点

因为 GAN 模型并没有对真实数据的分布做出任何假设，也没有对模型本身的形式加以太多约束，所以它的自由度很高，可以用于逼近任何概率分布，这是 GAN 模型区别于其他生成模型

最重要的特点。基于深度神经网络模型的生成器和判别器的表达能力很强，这使得 GAN 非常适合生成非结构化的高维数据，如高分辨率图像等。

与基于变分法的生成模型（如 VAE）相比，GAN 模型没有引入近似条件和额外假设，因此能够保证生成的效果更好。以图像生成为例，GAN 生成的图像通常比 VAE 生成的图像更加清晰。与基于马尔可夫链算法的生成模型相比，GAN 模型的训练过程不需要依赖马尔可夫链采样，因此计算复杂度不高，训练速度较快，能够生成数据的维度更高。另外，以 FVBN 为代表的一些生成模型必须串行地生成数据在各个维度上的值，计算速度受限，而 GAN 模型的训练和推理计算都很容易并行化，从而能够充分利用 GPU 等高性能计算设备加速计算。

可以看出，GAN 模型有很多优点。然而，原始的 GAN 模型也并不是完美无缺的。比如，在其目标函数中，当生成器生成的数据和真实数据的分布之间没有交集时，JS 散度为常数。此时如果判别器也为最优，就会出现梯度消失现象，继而难以根据该目标函数更新生成器模型的参数。另外，由于灵活性太高，GAN 模型经常会出现模式崩溃（mode collapse）问题，即生成器和判别器的参数优化过程停滞不前、无法收敛甚至退化。这使得 GAN 中生成器和判别器这两个矛盾体不易同时达到最优（即达到纳什均衡）。不过，最新的 GAN 模型变种，如 WGAN（Wasserstein GAN）、minibatch GAN 等已经较好地解决了这些问题。GAN 模型还在快速发展演进中，它在很多实际应用中的表现已经超越了绝大多数生成模型。

10.1.3　应用

在很多图像应用中，我们期望得到逼真的图像数据。GAN 模型的生成器和判别器之间的博弈使得生成器的输出无限逼近真实数据的分布，它可以使图像的生成效果更好。在半监督学习和强化学习等其他学习范式或者框架中，我们也可以利用 GAN 强大的数据生成能力获得更好的训练效果。此外，近年来 GAN 模型也开始应用于自然语言处理等其他领域。

- **图像翻译。**这是指文字到图像的转换，或不同风格图像之间的转换。文字到图像的转换是指根据一段文字生成其所描述的一幅图像。它包含两个方面，一是抽取文字所描述的语义特征，二是根据这些语义特征生成一幅相应的图像。在用于文字到图像转换的 GAN 模型中，生成器的输入包括噪声向量和文字的编码向量，这两个向量同时输入到生成器中用于图像生成，使得图像的语义与相应文字的语义产生关联。判别器不仅需要判断图像看起来是否真实，而且需要判断图像是否与文字编码向量相互对应。除了将文字转换为图像外，我们还可以将图像在不同风格之间进行转换。例如，输入一张手工画的草图，输出该草图语义所表达的真实的图像；或者，输入一张真实的照片，输出照片融入某位画家风格的图像。在图像风格转换时，我们需要保证语义的完整性，一般不会修改图像中的某些语义（如删除图像中的一个具体物体）。
- **图像编辑。**不同于不改变语义的图像翻译，图像编辑是指对图像中指定的某些内容进行语义上的修改。在GAN模型中，我们可以通过改变生成器输入向量中的某些维度的值来控制生成器输出图像的局部语义信息。比如对于人脸图像，可以通过改变输入向量中的

某些值来控制所生成人脸的脸型、发型、表情等。为了使用户可以编辑模型的隐藏向量，进而控制图像的语义信息，我们通常需要将编码器与GAN模型结合使用。例如，我们可以在训练完GAN模型之后，再训练两个编码器。这两个编码器分别将待编辑的图像转换为一个噪声向量和一个隐藏向量。隐藏向量的每个维度上的值都表示一些高级语义特性。我们将这两个向量同时作为GAN模型生成器的输入，就可以通过修改隐藏向量来控制 GAN 模型的生成器所生成图像的某些局部语义信息。

- **图像超分辨率**。图像超分辨率是计算机视觉和图像处理领域的一个重要研究方向，其核心目标是根据低分辨率图像生成对应的高分辨率图像。将深度学习应用于图像超分辨率时，模型的输入为低分辨率图像，输出为相应的高分辨率图像，损失函数为输出图像和真实高分辨率图像之间的像素级 MSE（Mean Square Error）。因为高分辨率的图像具有超高维度，而超高维空间中图像数据的分布非常稀疏，所以仅靠MSE训练出的模型所生成的图像在某些细节处看起来并不真实。由于GAN对真实数据的分布具有更好的拟合，它可以用于生成更加真实的高分辨率图像。在用于图像超分辨率的GAN模型中，生成器的输入为低分辨率图像，输出为高分辨率图像；判别器的输入为真实的高分辨率图像和生成的高分辨率图像。我们可以将图像超分辨率方面的经典损失函数引入到GAN模型的训练中，使得生成的图像在内容方面的信息丢失最小。
- **半监督学习**。半监督学习是指同时利用带标签的数据和不带标签的数据进行模型训练的机器学习方法。传统的深度学习模型特别依赖大量带标签的数据。然而在实际应用中，获得大量准确的带标签数据的成本往往很高。因此，我们通常采用半监督学习的方式来训练深度学习模型。GAN 模型的生成器能产生逼近真实的数据，这些数据可以作为不带标签的数据用于半监督学习。例如，对于一个K分类问题(样本所属类别数为$1, 2, \cdots, K$)，我们可以将GAN模型生成器产生的数据看作第K+1类，并将其作为训练数据的一部分。这样，原始的K分类问题转变为K+1分类问题，训练数据得到了扩充，这使得模型的训练更加容易。该分类模型的最优参数为$\boldsymbol{\theta}=\text{argmax}_{\boldsymbol{\theta}}V=\text{argmax}_{\boldsymbol{\theta}}(V_s+V_u)$，其中$V_s=E_{\boldsymbol{x},y\sim p_{\text{data}}(\boldsymbol{x},y)}[\log P(y|\boldsymbol{x}, y\leqslant K)]$和$V_u=E_{\boldsymbol{x},y\sim p_{\text{data}}(\boldsymbol{x},y)}[1-\log P(y=K+1|\boldsymbol{x})]+E_{\boldsymbol{x}\sim G}[\log P(y=K+1|\boldsymbol{x})]$分别表示生成数据（无标签的数据）和原始带标签的数据各自对应的目标函数。将这两类目标函数结合起来优化，可以实现半监督学习。
- **基于 GAN 的强化学习**。我们已经在 8.2 节中了解过强化学习。在强化学习中，有时我们需要对环境进行建模以使得智能体能够更好地与环境交互。GAN 模型可用于对环境建模，环境模型的输入是智能体的动作，输出是环境的状态信息。智能体在真正执行动作之前，可以先尝试给内部的环境模型输入不同的动作信号，选取能够获得最理想状态信息的动作。这样可以保证智能体在与环境真正交互的过程中执行的动作更加准确。另外，传统的强化学习大多属于单任务系统，智能体只能通过奖励函数一直执行一个单一的、不变的任务。OpenAI 的研究员 David Held 等人提出使用 GAN 为智能体持续地生成难度适宜的多个目标任务。这样智能体在与环境交互的过程中就能够学习到更多知识，不需要太多的环境先验知识就可以执行多个目标任务。

当前 GAN 的研究非常火热，很多新的应用也在不断出现，如图像去模糊等。类似于图像生成，GAN 模型还可以实现文本和对话生成等，这使得 GAN 可以用于自然语言处理领域。总体而言，GAN 模型作为一种通用的生成模型，可以与人工智能的很多研究方向结合，产生很多新型的应用。

10.2　GAN 模型的改进

自 2014 年 GAN 模型被首次提出之后，研究者们很快发现它在实际应用中存在很多问题，如生成图像的分辨率不高、学习的特征不可控、训练过程不稳定等。为了解决这些问题，研究者们对原始的 GAN 模型进行了持续不断的改进，比如修改或增加生成器和判别器的输入信息、重新设计生成器和判别器的模型架构、改进生成器和判别器的最优化目标函数等。下面我们介绍近两三年来 GAN 发展和演变过程中出现的一些改进版本。因为 GAN 模型在图像方面的应用相对成熟，所以大部分改进后的模型都首先用于解决图像类应用面临的问题。尽管如此，这些改进思想大多具有普适意义，可以较容易地迁移到其他应用中。同时，这些模型的设计思想可以相互结合，形成更强大的 GAN 模型。

10.2.1　CGAN模型

原始的 GAN 框架不需要对数据的分布做假设，而是直接采用多层神经网络模拟数据分布。这种训练方式太过自由，没有任何约束条件。在实际应用中，数据标签或其他辅助数据可被用作训练 GAN 的约束条件，这些约束条件有助于引导生成模型对数据分布的拟合。加拿大蒙特利尔大学的 Mehdi Mirza 等人提出了 CGAN（Conditional GAN，条件生成对抗网络）模型，该模型在判别器和生成器的输入中分别引入了一个条件向量 $\boldsymbol{y}$，用于约束生成图像的某些属性（如标签等）。CGAN 的原理框图如图 10-2 所示，其最优化问题可被描述为 $\min_G\max_D V_C(D,G)=E_{x\sim p_{\text{data}}}[\log D(\boldsymbol{x}|\boldsymbol{y})]+E_{z\sim p_z(z)}[\log(1-D(G(\boldsymbol{z}|\boldsymbol{y})))]$。

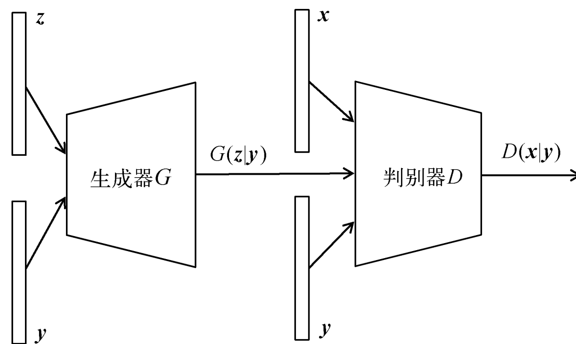


图 10-2　CGAN 的原理框图

我们以手写体数字生成模型为例说明 CGAN 的应用场景。当希望控制生成手写体数字的值

时，可以在生成器和判别器的输入中分别引入一个 One-Hot 编码的 10 维条件向量 $\boldsymbol{y}$ 作为标签约束。具体实现方式为：在生成器中，服从某一分布的 100 维噪声向量 $\boldsymbol{z}$ 和标签约束 $\boldsymbol{y}$ 分别经过一层隐藏层变换后，同时输入到下一隐藏层中；在判别器中，输入图像（真实图像 $\boldsymbol{x}$ 或生成图像 $G(\boldsymbol{z}|\boldsymbol{y})$和标签约束 $\boldsymbol{y}$ 也分别经过各自的隐藏层变换，然后合并后输入到 sigmoid 层。在此应用场景下，CGAN 模型中的条件向量还可用于表示其他属性，如生成的手写体数字的笔迹宽度等。

CGAN 模型通过引入条件向量，实现了对生成模型训练过程的引导。当模型训练完成后，我们可以通过改变条件向量中某些维度上的数值来控制生成的数据。

10.2.2 LAPGAN模型

原始的 GAN 模型只适用于生成分辨率较低的图像。为了提升生成图像的分辨率，纽约大学的 Emily Denton 等人提出了 LAPGAN（Laplacian Pyramid GAN，拉普拉斯金字塔生成式对抗网络）模型。该模型的主要特点是将图像的拉普拉斯金字塔同 GAN 结合起来，以多阶段图像生成的方式间接地实现高分辨率图像的生成。所谓图像金字塔，是指由原始图像（或原始图像的变换）及其所对应的一系列低分辨率图像组成的集合。如果将这一系列分辨率由低到高的图像从上到下排列，就形成了金字塔形状。高斯金字塔是一种经典的图像金字塔结构，其主要特征是：在每个图像降采样过程前都先进行高斯模糊。如图 10-3 所示，$\{\boldsymbol{I}_0,\cdots,\boldsymbol{I}_k,\cdots,\boldsymbol{I}_K\}$为图像的高斯金字塔，其中 $k=0,\cdots,K$ 代表不同的尺度。$\boldsymbol{I}_0$ 为原始图像，后续每幅图像 $\boldsymbol{I}_k$ 是由上一幅图像 $\boldsymbol{I}_{k-1}$ 经过高斯模糊和降采样后得到的，其大小为 $\boldsymbol{I}_{k-1}$ 的四分之一。拉普拉斯金字塔是用于描述高斯金字塔中丢失的高频信息的一种图像金字塔结构。在图 10-3 中，$\boldsymbol{I}_k$ 经过升采样后与 $\boldsymbol{I}_{k-1}$ 之间的差 $\boldsymbol{L}_{k-1}$ 表示拉普拉斯金字塔在第 $k-1$ 尺度下的图像，该图像代表 $\boldsymbol{I}_{k-1}$ 降采样过程中损失的高频信息。对于第 K 尺度，$\boldsymbol{L}_K=\boldsymbol{I}_K$。从 $\boldsymbol{L}_k$ 开始，由拉普拉斯金字塔可以重构出原始图像 $\boldsymbol{I}_0$。

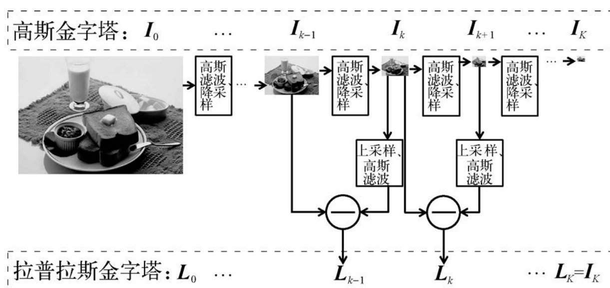


图 10-3 高斯金字塔和拉普拉斯金字塔示意图（另见彩插）

LAPGAN 的原理如图 10-4 所示。在 LAPGAN 中，每个尺度下都有一个生成器，用于生成该尺度下的拉普拉斯金字塔图像。例如，在 K 尺度下，噪声向量 $\boldsymbol{z}_K$ 输入到生成器 G_K 中得到生成图像 $\boldsymbol{J}_K$。在 $K-1$ 尺度下，生成器 G_{K-1} 的输入为噪声向量 $\boldsymbol{z}_{K-1}$，以及 $\boldsymbol{J}_K$ 升采样后得到的图像 $\boldsymbol{J}'_K$（类似于 CGAN 的做法，该图像作为条件变量引导 G_{K-1} 的输出）。根据拉普拉斯金字塔图像和高斯金字塔图像之间的关系，G_{K-1} 的输出与 $\boldsymbol{J}'_K$ 相加后即可得到 $K-1$ 尺度下的生成图像 $\boldsymbol{J}_{K-1}$。以此类推，

该模型最后通过生成器 G_0 得到最终生成的图像 $\boldsymbol{J}_0$。由此可以看出，LAPGAN 通过多组生成器逐渐实现高分辨率图像的生成。这本质上是一种逐步渐进的约束方式，它解决了原始 GAN 模型太过自由而难以控制的问题。

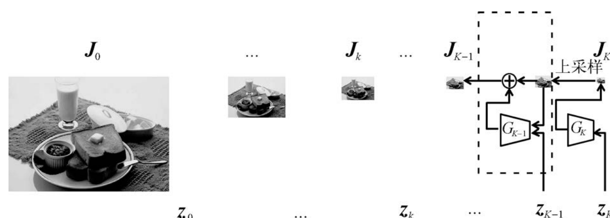


图 10-4　LAPGAN 的原理框图（另见彩插）

10.2.3　DCGAN模型

在原始的 GAN 模型中，生成器和判别器都是浅层模型。为了生成分辨率更高的图像，我们需要更深的模型。Indico 公司的 Alec Radford 等人于 2016 年提出了 DCGAN（Deep Convolutional GAN，深度卷积对抗生成网络）模型。该模型中判别器和生成器都采用全卷积神经网络。除了判别器的最后一层和生成器的第一层之外，其他层都采用卷积层。该模型提升了 GAN 生成大尺度图像的能力，它在 LSUN（Large-scale Scene Understanding）数据集上训练后可生成大小为 64×64 的逼真图像，超过了之前的其他 GAN 模型。

DCGAN 模型的整体架构与原始的 GAN 模型一致（如图 10-1 所示），区别在于生成器和判别器在具体实现方面的几个细节：首先，判别器中的池化层被带步长的卷积层（strided convolution）取代。这使得 CNN 能够自动学习特征图的降采样模式，而非按照固定的最大池化或平均池化做降采样。这种降采样方式也使得整个 DCGAN 模型变为完全可微的，对于训练的稳定性有好处。其次，判别器（除了输出层之外）和生成器（除了输入层之外）的卷积层输出结果都经过 BN 层做归一化。基于 BN 层的归一化使得训练精度对于参数初始化不敏感，并且能够加快训练速度。

另外，为了使得生成的图像数据更逼真，DCGAN 模型的设计也包含一些小技巧，比如生成器（除了输出层之外）主要采用 ReLU 作为激活函数，判别器主要采用 Leaky ReLU 作为激活函数；训练时的优化器选用 Adam 而不是带动量的 SGD。以上这些特性虽然没有严格的数学证明，但是会使得生成大尺度图像的效果非常不错。

10.2.4　InfoGAN模型

原始的 GAN 模型训练出的生成器，其输入向量的每个维度不代表具体的语义信息，输出图像的语义通常隐藏地分布在输入向量的各个维度中。假设生成器的输出图像为人脸，当我们想编辑图像的某个语义信息（如眼睛颜色）时，需要进一步挖掘输入向量中的维度与图像语义的映射关系，并确保编辑过程只作用于特定的图像语义。为了使生成器的输入向量的一些维度具有明确

的语义，OpenAI 公司的 Xi Chen 等人提出 InfoGAN 模型。

在 InfoGAN 模型中，生成器的输入向量由两部分构成，一部分是不可压缩的噪声向量 $\boldsymbol{z}$，另一部分是可代表明显语义特征的向量 $\boldsymbol{c}$。向量 $\boldsymbol{c}$ 中每个维度的值可以是离散的（比如手写体数字的类别），也可以是连续的（比如手写体数字的倾斜度）。我们希望向量 $\boldsymbol{c}$ 和生成器输出图像之间的互信息最大，以使得向量 $\boldsymbol{c}$ 中每个维度都代表明确的语义信息。因此，InfoGAN 模型的最优化问题可被描述为 $\min_G\max_D V_I(D,G)=\min_G\max_D(V(D,G)-\lambda I(\boldsymbol{c};G(\boldsymbol{z},\boldsymbol{c})))$。其中，$V(D,G)=E_{\boldsymbol{x}\sim p_{\text{data}}}[\log D(\boldsymbol{x})]+E_{z\sim p_z(z)}[\log(1-D(G(\boldsymbol{z})))]$ 是标准 GAN 模型的损失函数，$I(\boldsymbol{c};G(\boldsymbol{z},\boldsymbol{c}))$表示向量 $\boldsymbol{c}$ 与生成器输出图像 $G(\boldsymbol{z},\boldsymbol{c})$之间的互信息，互信息越大说明相关性越高。根据信息熵的定义，$I(\boldsymbol{c};G(\boldsymbol{z},\boldsymbol{c}))$可进一步表示为 $I(\boldsymbol{c};G(\boldsymbol{z},\boldsymbol{c}))=H(\boldsymbol{c})-H(\boldsymbol{c}|G(\boldsymbol{z},\boldsymbol{c}))=E_{\boldsymbol{x}\sim G(z,c)}[E_{c'\sim P(c|x)}[\log P(\boldsymbol{c}'|\boldsymbol{x})]]+H(\boldsymbol{c})$，其中 $H(\cdot)$ 表示信息熵。直接优化 $I(\boldsymbol{c};G(\boldsymbol{z},\boldsymbol{c}))$ 意味着需要对隐藏变量 $\boldsymbol{c}$ 的后验概率分布 $P(\boldsymbol{c}|\boldsymbol{x})$ 做采样。如果采用蒙特卡洛模拟法对后验证概率分布 $P(\boldsymbol{c}|\boldsymbol{x})$ 做近似计算，则计算量很大，速度比较慢，而且有一定随机性。为此，InfoGAN 通过引入变分法解决这个问题。它采用一个辅助分布 $Q(\boldsymbol{c}|\boldsymbol{x})$ 来近似 $P(\boldsymbol{c}|\boldsymbol{x})$，从而避免对 $P(\boldsymbol{c}|\boldsymbol{x})$直接采样，简化了计算。具体公式为 $I(\boldsymbol{c};G(\boldsymbol{z},\boldsymbol{c}))=E_{\boldsymbol{x}\sim G(z,c)}[D_{KL}(P(\cdot|\boldsymbol{x})||Q(\cdot|\boldsymbol{x}))+E_{c'\sim P(c|x)}[\log Q(\boldsymbol{c}'|\boldsymbol{x})]]+H(\boldsymbol{c})$。其中，$D_{KL}(P(\cdot|\boldsymbol{x})||Q(\cdot|\boldsymbol{x}))$表示真实的后验概率分布与其近似分布之间的 KL 散度。由于 KL 散度的非负性，我们可以得到 $I(\boldsymbol{c};G(z,\boldsymbol{c}))\geqslant L_I(G,Q)=E_{\boldsymbol{x}\sim G(z,x)}[E_{c'\sim P(c|x)}[\log Q(\boldsymbol{c}'|\boldsymbol{x})]]+H(\boldsymbol{c})$。$I(\boldsymbol{c};G(\boldsymbol{z},\boldsymbol{c}))$ 的最大化问题可转变为 $L_I(G,Q)$ 的最大化问题。在实际中，我们可以采用深度学习模型表示 $Q(\boldsymbol{c}|\boldsymbol{x})$。InfoGAN 的原理框图如图 10-5 所示。可以看出，InfoGAN 在原始的 GAN 基础上引入了向量 $\boldsymbol{c}$ 和辅助分布 $Q(\boldsymbol{c}|\boldsymbol{x})$，辅助分布与判别器共享输出层以外的其他层。

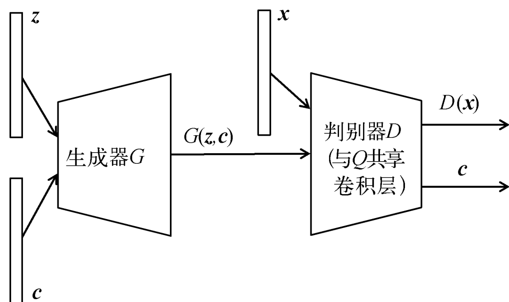


图 10-5 InfoGAN 的原理框图

10.2.5 LSGAN模型

上面几种 GAN 模型的目标函数大部分基于 sigmoid 交叉熵，这种目标函数有时会导致梯度消失问题。当生成器 G 生成的数据处于判别器 D 的分类面的正确一侧，但又离真实数据分布比较远时（比如生成的图像质量还不高时），我们很难通过该目标函数计算出相应的梯度更新生成器 G 的模型参数。生成器 G 将难以被继续优化，此时出现了梯度消失现象。LSGAN（Least Suqares GAN，最小二乘 GAN）模型是一种采用基于最小二乘法的目标函数分别优化生成器和判别器的 GAN 模型。

为了使判别器 D 对生成器 G 生成的数据做正确的分类，同时使生成器生成的数据与真实数据在流形空间中尽可能靠近，LSGAN 模型的目标优化函数被设计为 $\min_D V_{\text{LSGAN}}(D)=\frac{1}{2}(E_{\boldsymbol{x}\sim p_{\text{data}}}\left[(D(\boldsymbol{x})-b)^2\right]+E_{z\sim p_z(z)}\left[(D(G(\boldsymbol{z}))-a)^2\right])$ 和 $\min_G V_{\text{LSGAN}}(G)=\frac{1}{2}E_{z\sim p_z(z)}\left[(D(G(\boldsymbol{z}))-c)^2\right]$。其中，$a$、$b$ 分别为生成数据和真实数据的标签，c 是生成器 G 为了“迷惑”判别器 D 而给生成数据设定的标签。LSGAN 模型的整体架构与原始的 GAN 模型基本一致，主要的不同点在于目标优化函数。当固定生成器 G 时，对判别器 D 的目标优化函数求导可得出最优的判别器：$D^*=\frac{bp_{\text{data}}(\boldsymbol{x})+ap_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x})+p_g(\boldsymbol{x})}$。当判别器 $D=D^*$ 时，如果 $b-c=1$ 且 $b-a=2$，则 $V_{\text{LSGAN}}(G)=\frac{1}{2}\times\chi^2_{\text{Pearson}}((p_{\text{data}}+p_g)\|2p_g)$。其中，$\chi^2_{\text{Pearson}}(\cdot)$ 为皮尔森卡方散度。它类似于 KL 散度，可以用于描述两个概率分布之间的相似度。为了满足 $b-c=1$ 和 $b-a=2$，一般 a、b、c 的取值分别为−1、1、0。另外，基于 sigmoid 交叉熵的目标函数容易出现饱和，从而导致无法继续训练。但最小二乘法目标函数则不容易出现饱和现象，这使得训练过程变得更加容易。实验效果表明，相比原始的 GAN 模型，LSGAN 模型生成的图像分辨率更高，且训练过程更加稳定。

10.2.6　WGAN模型

如 10.1.2 节所述，在常规 GAN 模型的目标优化函数中，使用 JS 散度可能引发梯度消失现象而使得生成器无法更新。为了避免这一问题，纽约大学的 Martin Arjovsky 等人提出了 WGAN 模型，采用地动距离（EMD，Earth Mover’s Distance）来定义生成数据和真实数据分布之间的差异。地动距离被定义为 $W(p_{\text{data}}\|p_g)=\inf_{\gamma\in\Pi(p_{\text{data}},p_g)}E_{(\boldsymbol{x},\boldsymbol{y})\sim\gamma}\left[\|\boldsymbol{x}-\boldsymbol{y}\|\right]$。其中，符号 inf 表示函数的下界，$\gamma$ 表示由边缘概率分布 p_{data} 和 p_g 共同构成的联合概率分布。地动距离表示将一个概率分布转换为另一个概率分布所需经过的最短距离。相比于 JS 散度，它可以做到处处连续且可微。当两个分布之间没有任何交集时，地动距离也可以给出有效的度量，这样可以解决原始 GAN 模型中判别器处于最优状态时生成器的梯度消失问题。然而，直接求取地动距离的下界在数学上是很难的。为此，Martin Arjovsky 等人提出使用 Kantorovich-Rubinstein 对偶将原问题进行变换。这个变换的前提是判别器为一阶的利普希茨连续函数。利普希茨连续性要求函数的导数在所有位置上的绝对值都不能超过一个既定的常数。为方便起见，早期版本的 WGAN 模型采用了简单的权重裁剪（weight clipping）方法。当梯度过大导致模型中每层权重参数过大时，该方法将权重值限定在一定取值范围内。权重裁剪会给 GAN 模型添加一个超参选项，即裁剪的阈值 c。虽然权重裁剪实现起来很简单，但是 c 这个超参如果设置得不好，就会在判别器中引发梯度消失或梯度爆炸。另外，由于取值范围的限定，判别器的参数分布容易出现极端化，会有大量的参数集中在 $-c$ 和 c 这两个值附近。改进后的 WGAN 模型采用了梯度惩罚（gradient penalty）方法以使判别器满足利普希茨连续性条件，具体做法是在判别器的损失函数中添加一个正则项来限定权重梯度的取值范围。

在 WGAN 模型的实际应用中，Martin Arjovsky 等人也提出了一些有用的小技巧，这些技巧已经在很多开源数据集上得到了验证。比如，SGD 和 RMSProp 等优化器通常表现较好，而 Adam 或者 Momentum SGD 等带有动量的优化器则表现不好，不能保证梯度更新的稳定性。另外，WGAN 模型中判别器地动距离近似值的大小与生成器生成图像的质量有强相关性。我们在用 WGAN 做实验时，可以根据地动距离近似值的大小直接估计出当前图像质量的好坏。

10.3　最佳实践

在基于 GAN 改进的模型中，DCGAN 比较有代表性，它很好地体现了深度卷积模型与 GAN 的结合。本节以该模型为例，介绍如何利用 TensorFlow 实现 GAN 模型。本节引用的代码来自 GitHub 社区的 carpedm20/DCGAN-tensorflow 项目。在所有与 DCGAN 实现相关的项目中，该项目的受欢迎程度最高。

该项目采用了模块化设计，将数据、算子、模型和训练程序的定义分开。这使得整个项目的代码结构简单清晰，没有复杂的层次关系。该项目主体代码由 download.py、utils.py、model.py、ops.py 和 main.py 这几个 Python 模块构成，每个模块的具体功能如下。

- download.py：负责下载和解压三个开源数据集（Celeb-A、LSUN 和 MNIST）。
- utils.py：提供通用的工具方法，涉及对图像执行打开、保存、预处理和可视化，以及对模型执行格式转换（如将模型转存为 JSON 格式）。
- ops.py：负责封装 DCGAN 模型用到的主要算子，包括 BN、Convolution、Deconvolution、Leaky ReLU、Linear（全连接）等，这些算子是对 TensorFlow Python API 的进一步封装，具体实现依赖于 tensorflow.nn 模块。
- model.py：用于定义包括生成器和判别器在内的 DCGAN 模型及其训练步骤。
- main.py：整个训练代码的入口程序，它读取用户设置的超参数，创建 `DCGAN` 对象，然后调用 `DCGAN` 对象的 `train` 方法启动训练。

可以看出，`model` 模块是整个项目的核心。下面我们首先介绍如何根据 `generator`、`discriminator` 等方法构造出 DCGAN 模型，然后以 `DCGAN` 类的 `train` 方法为主线，说明模型的训练过程。

`DCGAN` 类的初始化函数主要用于定义模型超参数，以及基于这些超参数建立模型。具体代码如下：

```
class DCGAN(object):
  def __init__(self, sess, input_height=108, input_width=108, crop=True,
         batch_size=64, sample_num=64, output_height=64, output_width=64,
         y_dim=None, z_dim=100, gf_dim=64, df_dim=64,
         gfc_dim=1024, dfc_dim=1024, c_dim=3, dataset_name='default',
         input_fname_pattern='*.jpg', checkpoint_dir=None, sample_dir=None):
    self.sess = sess # 当前会话
    self.crop = crop # 是否需要裁剪判别器的输入图像，即改变原图像高和宽
```

10

```
        self.batch_size = batch_size # 训练所用的批大小
        self.sample_num = sample_num # 每次迭代中生成器采样的样本数
        self.input_height = input_height # 训练集中输入图像的宽度
        self.input_width = input_width # 训练集中输入图像的高度
        self.output_height = output_height # 生成器生成的图像高度
        self.output_width = output_width # 生成器生成的图像宽度
        self.y_dim = y_dim # 生成器或判别器的输入条件向量维度
        self.z_dim = z_dim # 生成器的输入噪声向量维度
        self.gf_dim = gf_dim # 生成器的第一个卷积层的输出通道数
        self.df_dim = df_dim # 判别器的第一个卷积层的输出通道数
        self.gfc_dim = gfc_dim # 生成器的全连接层的输出神经元个数
        self.dfc_dim = dfc_dim # 判别器的全连接层的输出神经元个数
        # ...
        # 定义生成器和判别器的 BN 层
        # ...
        self.dataset_name = dataset_name # 训练数据集的名称
        # ...
        # 定义输入图像的格式和通道数 c.dim（灰度图像的通道数为 1，彩色图像的通道数为 3）
        # ...
        self.build_model() # 建立模型
```

初始化函数调用的 `build_model` 成员方法用于构造整个 DCGAN 模型，其基本流程为：首先，新建一个生成器（generator）、一个判别器（distriminator）和一个采样器（sampler）对象。采样器的模型参数来自于生成器的模型参数，可用于对模型的好坏作出评价。这里可以选择性地将 CGAN 的设计思想与 DCGAN 结合起来，具体实现方式是将条件向量 $\boldsymbol{y}$ 加入到生成器和判别器对象的输入中。然后，根据 GAN 模型的原理构造判别器和生成器的损失函数，以此建立起二者互相博弈的规则。最后，将生成器和判别器的模型参数分别用单独的张量保存引用，便于做更新、保存等操作。该方法的具体代码如下：

```
    def build_model(self):
      if self.y_dim:
        # 定义生成器或者判别器的输入条件向量
        self.y = tf.placeholder(tf.float32, [self.batch_size, self.y_dim], name='y')
      else:
        self.y = None
      # 如果 self.crop 为 True，则在输入图像的中央裁剪出高度和宽度分别为
      # self.output_height 和 self.output_width 大小的图像
      if self.crop:
        image_dims = [self.output_height, self.output_width, self.c_dim]
      else:
        image_dims = [self.input_height, self.input_width, self.c_dim]
      # 定义判别器的输入数据，即真实的图像
      self.inputs = tf.placeholder(tf.float32, [self.batch_size] + image_dims, name='real_images')
      inputs = self.inputs
      self.z = tf.placeholder(tf.float32, [None, self.z_dim], name='z')
      # ...
      # 分别创建生成器、判别器和采样器，并得到相应的输出张量。在训练过程中，
      # 每隔一定的迭代步数，采样器被调用一次，其本质是用生成器做一次推理并得到生成的图像，
      # 通过对生成的图像做质量评价，我们可以知道当前生成器的好坏
      self.G = self.generator(self.z, self.y)
      self.D, self.D_logits = self.discriminator(inputs, self.y, reuse=False)
```

```
    self.sampler = self.sampler(self.z, self.y)
    self.D_, self.D_logits_ = self.discriminator(self.G, self.y, reuse=True)
    # ...
    # 定义判别器在真实数据分布上的损失函数
    self.d_loss_real = tf.reduce_mean(
        sigmoid_cross_entropy_with_logits(self.D_logits, tf.ones_like(self.D)))
    # 定义判别器在生成数据分布上的损失函数
    self.d_loss_fake = tf.reduce_mean(
        sigmoid_cross_entropy_with_logits(self.D_logits_, tf.zeros_like(self.D_)))
    # 定义生成器在生成数据分布上的损失函数
    self.g_loss = tf.reduce_mean(
        sigmoid_cross_entropy_with_logits(self.D_logits_, tf.ones_like(self.D_)))
    # ...
    self.d_loss = self.d_loss_real + self.d_loss_fake # 判别器总的损失函数
    t_vars = tf.trainable_variables() # 所有可被训练的参数
    self.d_vars = [var for var in t_vars if 'd_' in var.name] # 判别器中可被训练的参数
    self.g_vars = [var for var in t_vars if 'g_' in var.name] # 生成器中可被训练的参数
    # ...
```

需要注意的是，当训练数据为 MNIST 数据集时，DCGAN-tensorflow 项目默认采用数据集的图像标签作为条件向量对生成器（或判别器）做约束。因为 MNIST 数据集中图像分辨率较低、数据复杂度不高，所以生成器（或判别器）也可以构造得简单一些，两层反卷积（或卷积）即可满足特征提取的要求。对于其他数据集，该项目主要采用四层反卷积（或卷积）来构造生成器（或判别器），以充分提取图像数据的特征，此时默认不需要条件向量的约束。

依据条件向量 $\boldsymbol{y}$ 的有无，生成器的基本工作流程具有两个分支。generator 函数的具体代码如下：

```
def generator(self, z, y=None): # 生成器
  with tf.variable_scope("generator") as scope:
    if not self.y_dim:
      # 将输出图像的高度和宽度分别设定为 self.output_height 和 self.output_width。
      # 因为每层反卷积操作的 stride 都设为 2，所以每层反卷积之后图像的高度和宽度都会加倍
      s_h, s_w = self.output_height, self.output_width
      s_h2, s_w2 = conv_out_size_same(s_h, 2), conv_out_size_same(s_w, 2)
      s_h4, s_w4 = conv_out_size_same(s_h2, 2), conv_out_size_same(s_w2, 2)
      s_h8, s_w8 = conv_out_size_same(s_h4, 2), conv_out_size_same(s_w4, 2)
      s_h16, s_w16 = conv_out_size_same(s_h8, 2), conv_out_size_same(s_w8, 2)
      # 将生成器的输入噪声向量通过线性变换映射到 self.gf_dim*8*s_h16*s_w16 维度的向量
      self.z_, self.h0_w, self.h0_b = linear(z, self.gf_dim*8*s_h16*s_w16,
                                             'g_h0_lin', with_w=True)
      self.h0 = tf.reshape(self.z_, [-1, s_h16, s_w16, self.gf_dim * 8])
      h0 = tf.nn.relu(self.g_bn0(self.h0))
      # 按照调参经验，除了最后一层激活函数采用 tanh 之外，其他层激活函数都采用 ReLU
      self.h1, self.h1_w, self.h1_b = deconv2d(
          h0, [self.batch_size, s_h8, s_w8, self.gf_dim*4], name='g_h1', with_w=True)
      h1 = tf.nn.relu(self.g_bn1(self.h1))
      # ...
      # 类似于 h1 的求取过程，通过连续调用反卷积操作（deconv2d）、
      # BN 操作（self.g_bn1、self.g_bn2、self.g_bn3）和激活操作（tf.nn.relu），得到 h2 和 h3
      # ...
      h4, self.h4_w, self.h4_b = deconv2d(
          h3, [self.batch_size, s_h, s_w, self.c_dim], name='g_h4', with_w=True)
```

10

```
            return tf.nn.tanh(h4)
        else:
            # 当有条件向量 y 时，构造一个主要由两层卷积和两层全连接组成的相对简单的模型
            # 因为有条件向量 y，所以将每个激活层的输出向量与条件向量 y 连接起来，以形成条件约束
            # ...
```

采样器 sampler 使用与生成器 generator 相同的模型架构及模型参数，因此其代码实现不再赘述。二者的区别在于：在实际训练过程中，采样器仅在每隔一定步数后才执行一次前向传播计算，而生成器则在每一步都要执行一次前向传播计算和一次后向传播计算。

类似于生成器 generator，判别器 distriminator 的基本工作流程也具有两个分支：当没有条件向量时，判别器主要由四个卷积层和一个全连接层组成；当有条件向量时，判别器主要由两个全连接层和两个卷积层组成。为了保证条件向量对判别器的影响，判别器中每个激活层之后的数据都会与条件向量相拼接。distriminator 函数的具体代码如下：

```
def discriminator(self, image, y=None, reuse=False): # 判别器
  with tf.variable_scope("discriminator") as scope:
    # 复用已有变量
    if reuse:
      scope.reuse_variables()
    # 如果没有条件向量 y，则建立一个带有四层卷积操作的卷积模型。为了避免梯度消失现象，
    # 每层卷积之后都加 BN 层。除了最后一层激活函数为 sigmoid 之外，其他层激活函数为 leaky ReLU
    if not self.y_dim:
      h0 = lrelu(conv2d(image, self.df_dim, name='d_h0_conv'))
      h1 = lrelu(self.d_bn1(conv2d(h0, self.df_dim*2, name='d_h1_conv')))
      h2 = lrelu(self.d_bn2(conv2d(h1, self.df_dim*4, name='d_h2_conv')))
      h3 = lrelu(self.d_bn3(conv2d(h2, self.df_dim*8, name='d_h3_conv')))
      h4 = linear(tf.reshape(h3, [self.batch_size, -1]), 1, 'd_h4_lin')
      return tf.nn.sigmoid(h4), h4
    else:
      # 与 not self.y_dim 分支中建模的特点相同，除最后一层外的每一层都包含
      # BN、leaky ReLU 这两个算子，区别在于后两层不是卷积层，而是全连接层
      # ...
```

DCGAN 对象实例化之后，调用其 train 方法即可开始训练。train 方法的基本流程为：首先通过最小化损失函数分别定义生成器和判别器的优化操作，用于在每次迭代过程中更新各自的模型参数；然后初始化生成器和判别器的模型参数，并定义用于 sampler 计算的数据；最后循环执行指定次数的训练过程。具体代码如下：

```
def train(self, config):
  # 调用 Adam 优化器的 minimize 方法得到生成器和判别器的优化操作
  d_optim = tf.train.AdamOptimizer(config.learning_rate, beta1=config.beta1) \
            .minimize(self.d_loss, var_list=self.d_vars)
  g_optim = tf.train.AdamOptimizer(config.learning_rate, beta1=config.beta1) \
            .minimize(self.g_loss, var_list=self.g_vars)
  # 初始化当前会话中所有的变量
  try:
    tf.global_variables_initializer().run()
  except:
    tf.initialize_all_variables().run()
  # ...
```

```
# 定义采样数据，用于 sampler 的计算
# ...
counter = 1 # 迭代步数的计数器
# ...
# 根据 config 中指定的 epoch 总数（默认为 25）开始循环执行训练
for epoch in xrange(config.epoch):
  # ...
  for idx in xrange(0, batch_idxs):
    # 读取批数据用于单步训练
    if config.dataset == 'mnist':
      batch_images = self.data_X[idx*config.batch_size:(idx+1)*config.batch_size]
      batch_labels = self.data_y[idx*config.batch_size:(idx+1)*config.batch_size]
    else:
      batch_files = self.data[idx*config.batch_size:(idx+1)*config.batch_size]
      # 调用 utils.py 提供的 get_image 方法调整图像大小，图像的高度由 self.input_height
      # 变为 self.output_height，宽度由 self.input_width 变为 self.output_width
      batch = [get_image(batch_file,
                input_height=self.input_height,
                input_width=self.input_width,
                resize_height=self.output_height,
                resize_width=self.output_width,
                crop=self.crop,
                grayscale=self.grayscale) for batch_file in batch_files]
    # ...
    batch_z = np.random.uniform(-1, 1, [config.batch_size, self.z_dim]).astype(np.float32)
    # 对于 MNIST 数据集，执行特殊的处理逻辑
    if config.dataset == 'mnist':
      # 执行 d_optim 操作，更新判别器的模型参数
      _, summary_str = self.sess.run([d_optim, self.d_sum], feed_dict={self.inputs:
                                                                        batch_images,
                                                                        self.z: batch_z,
                                                                        self.y:batch_
                                                                        labels,})
      # ...
      # 执行 g_optim 操作，更新生成器的模型参数
      _, summary_str = self.sess.run([g_optim, self.g_sum], feed_dict={self.z: batch_z,
                                                                        self.y:batch_
                                                                        labels,})
      # ...
    else:
      # 对于其他数据集，同样依次执行 d_optim 和 g_optim 操作，
      # 该分支与 mnist 分支代码的区别在于此处的判别器和生成器没有条件约束
      # ...
    counter += 1
    # ...
```

综上，DCGAN-tensorflow 项目以模块化的思路呈现了一个 GAN 模型的完整训练过程。我们可以在此基础上很方便地引入其他算子，实现自定义的判别器和生成器模型，以适应更复杂的数据集。另外，感兴趣的读者可以基于该项目做进一步的优化，比如可以用队列机制将模型训练时的数据读取变为异步操作，从而掩盖 IO 时间。

10.4 小结

GAN 是一种基于深度神经网络的生成模型。它由生成器和判别器构成，其训练过程即为二者互相博弈的过程。GAN 模型最大的好处在于灵活性，即不对数据的分布做任何假设，故而自由度非常大。GAN 模型巧妙地借助了深度学习模型（CNN 等）对数据分布的强大建模能力，使得生成逼真的图像等高维数据成为可能。然而，任何灵活性都是有代价的。原始 GAN 模型最大的问题就是约束条件太少，训练不稳定，容易出现模式崩溃，难以达到一个最优状态。针对这些问题，人们提出了诸多改进的 GAN 模型，如 CGAN、LAPGAN、DCGAN、InfoGAN、LSGAN、WGAN 等，这些模型使得生成的图像分辨率更高、训练更稳定。其中，DCGAN 是一种经典且常用的 GAN 模型。本章以 GitHub 社区的 carpedm20/DCGAN-tensorflow 项目为例，介绍了如何在 TensorFlow 中以模块化的形式实现 DCGAN。GAN 模型的应用场景很多，包括图像翻译、图像超分辨率、图像编辑、自然语言处理等，它还可以被应用到无监督学习和强化学习框架中。GAN 为机器学习和深度学习领域带来了一个新的研究方向，相信未来这方面会有更多突破性的进展。

第 11 章

RNN 模型

11

现实中的很多数据具有时间、空间等方面的序列特征，数据序列的前后部分具有逻辑上的关联性。神经网络的研究者们针对这类数据的语义感知与分析问题，设计了 RNN（循环神经网络）模型。这种模型具有时序上的“记忆”能力，因而在自然语言理解、语音识别等领域取得了显著效果。TensorFlow 的灵活架构为 RNN 模型的构建与训练提供了良好的支持。本章首先介绍常用的 RNN 单元及其对应的 TensorFlow 接口与实现，然后讲解如何使用 TensorFlow 开发基于 RNN 的语言模型和 Seq2Seq 模型。

11.1　基本 RNN 单元及其变种

RNN 模型是一种层内神经元之间形成有向环连接的神经网络模型。RNN 模型不属于前馈神经网络模型，它的主要特点在于隐藏层的输出不仅连接到下一层，而且还连接到自身。这种连接到自身的特点构成了数据的循环。通常，RNN 模型的输入和输出都具有“时序”特性。为了表达 RNN 模型的“循环”特性，可以将其按照输入序列的长度展开。展开后，RNN 模型可以看作由一系列核心模块（RNN 单元）形成的二维阵列。在该阵列中，前一时刻与后一时刻的 RNN 单元相互连接，上一层与下一层的 RNN 单元相互连接。在 RNN 单元内部，我们可以设计出各种复杂的计算流程，从而构造出不同的模型，满足不同场景的需求。本节介绍 RNN 模型的基本构成以及其中常用的几种单元。

11.1.1　RNN模型简介

在介绍基本 RNN 单元之前，我们先介绍一个简单的 RNN 模型。该模型的基本原理如图 11-1 所示，其中第 l 层的输入 $\boldsymbol{x}_0^l,\cdots,\boldsymbol{x}_t^l,\cdots,\boldsymbol{x}_T^l$ 为该层从 0 到 T 时刻的连续输入序列。该模型的计算逻辑为：在第 l 层，t 时刻的输入向量 $\boldsymbol{x}_t^l$ 与矩阵 $\boldsymbol{U}^l$ 相乘后的结果加上前一时刻该层的输出向量 $\boldsymbol{s}_{t-1}^l$ 与矩阵 $\boldsymbol{W}^l$ 相乘后的结果，再经过激活函数得到该层在 t 时刻的输出向量 $\boldsymbol{s}_t^l$ 。该输出向量 $\boldsymbol{s}_t^l$ 可以输入到下一层继续处理，也可以直接接入输出层，即与矩阵 $\boldsymbol{V}^l$ 相乘后再经过激活函数得到最终输出向量 $\boldsymbol{y}_t^l$ 。所有时刻的输出向量共同组成了该 RNN 模型输出序列 $\boldsymbol{y}_0^l,\cdots,\boldsymbol{y}_t^l,\cdots,\boldsymbol{y}_T^l$ 。该模型中矩阵 $\boldsymbol{U}^l,\boldsymbol{W}^l,\boldsymbol{V}^l$ 分别与各自输入向量做乘积，实现对输入向量的线性变换。

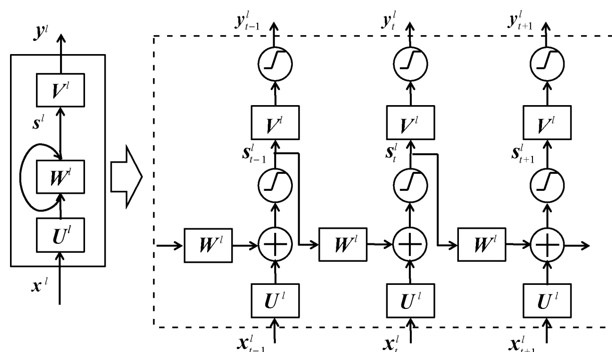


图 11-1　RNN 模型的基本原理图

RNN 模型中常用的单元包括基本 RNN 单元、LSTM 单元、GRU 单元、双向 RNN 单元和具有其他特性的 RNN 单元。这些单元类似于 CNN 模型中不同类型的层，是 RNN 模型的基本组成部分。但是由于多种连接的存在，这些单元的内部结构要比 CNN 中的层复杂很多。

11.1.2　基本RNN单元

上述 RNN 模型中用到的基本 RNN 单元如图 11-2 所示。该单元的输入为前一时刻隐藏层的状态向量 s_{t-1}^l 和当前时刻的输入向量 x_t^l，输出为当前时刻的隐藏层状态向量 s_t^l。该单元的输出向量既要输入到下一层同一时刻的基本 RNN 单元，也要输入到同一层下一个时刻的基本 RNN 单元。通过不同时刻隐藏层的状态传递，RNN 模型实现了对具有时间序列特性的输入数据的建模。

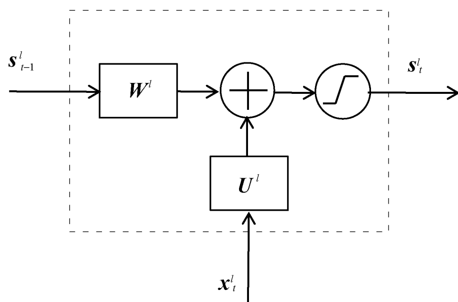


图 11-2　RNN 单元的内部结构图

在 TensorFlow 中，tensorflow/python/ops/rnn_cell_impl.py 文件定义的 `RNNCell` 类是一个抽象类，它是 RNN 单元及其变种的父类。继承 `RNNCell` 类的子类有 `BasicRNNCell`、`BasicLSTMCell`、`GRUCell` 和 `LSTMCell`。这些子类都有各自的调用方法，以实现类似于图 11-2 所示的虚线框内的计算逻辑。

在 `RNNCell` 类中，`call` 方法的输入参数为 `input` 和 `state`，维度分别为 `batch_size`×`input_size` 和 `batch_size`×`state_size`，它们分别表示批量的输入向量和前一时刻隐藏层状态向量。

输出参数为 output 和 next_state，维度分别为 batch_size×output_size 和 batch_size×state_size，它们分别表示批量的输出向量和当前时刻隐藏层状态向量。

RNNCell 类使用 @property 装饰器装饰 state_size 和 output_size 方法。其子类对象可以以属性的方式调用这两个方法，以获取隐藏层状态向量和输出向量的维度。

RNNCell 类的 _rnn_get_variable 方法用于将输入的 variable 划分为可训练参数和不可训练参数，并分别保存于该类的成员变量 _trainable_weights 和 _non_trainable_weights 中，相应的代码为：

```
def _rnn_get_variable(self, getter, *args, **kwargs):
  variable = getter(*args, **kwargs)
  trainable = (variable in tf_variables.trainable_variables() or
               (isinstance(variable, tf_variables.PartitionedVariable) and
                list(variable)[0] in tf_variables.trainable_variables()))
  if trainable and variable not in self._trainable_weights:
    self._trainable_weights.append(variable)
  elif not trainable and variable not in self._non_trainable_weights:
    self._non_trainable_weights.append(variable)
  return variable
```

RNNCell 通过 zero_state 方法返回值为 0 的隐藏层状态向量：

```
def zero_state(self, batch_size, dtype):
  with ops.name_scope(type(self).__name__ + "ZeroState", values=[batch_size]):
    state_size = self.state_size
    return _zero_state_tensors(state_size, batch_size, dtype)
```

在 TensorFlow 中，基本 RNN 单元由 tensorflow/python/ops/rnn_cell_impl.py 中的 BasicRNNCell 类实现，相应的代码为：

```
class BasicRNNCell(RNNCell):
  def __init__(self, num_units, activation=None, reuse=None):
    super(BasicRNNCell, self).__init__(_reuse=reuse)
    self._num_units = num_units
    self._activation = activation or math_ops.tanh
  @property
  def state_size(self):
    return self._num_units
  @property
  def output_size(self):
    return self._num_units
  def call(self, inputs, state):
    output = self._activation(_linear([inputs, state], self._num_units, True))
    return output, output
```

该类的 call 方法调用了 _linear 方法，将批量的输入向量 inputs 和前一时刻隐藏层状态向量 state 分别与各自对应的参数矩阵相乘后再相加得到中间结果，并可以选择是否添加偏置向量。_linear 方法得到的中间结果经过 _activation 方法后得到输出向量（或当前时刻的隐藏层状态向量）。在 _activation 方法中，激活函数一般选择 tanh。该 call 方法的输出为两个相同的 output，其中一个作为输入向量传入下一层的基本 RNN 单元，一个作为隐藏层状态传入当前层的下一个基本 RNN 单元。

11.1.3　LSTM单元

虽然基本 RNN 单元可以对具有时间序列的输入数据建模，但是当输入时间序列长度太长时，RNN 模型中较早时刻的隐藏层状态很难一直传递下去，这会造成信息丢失，出现长时间依赖问题（Long-Term Dependency）。为了解决这个问题，Sepp Hochreiter 和 Jürgen Schmidhuber 于 1997 年提出了 LSTM（Long Short-Term Memory，长短期记忆）单元，其内部详细结构如图 11-3 所示。

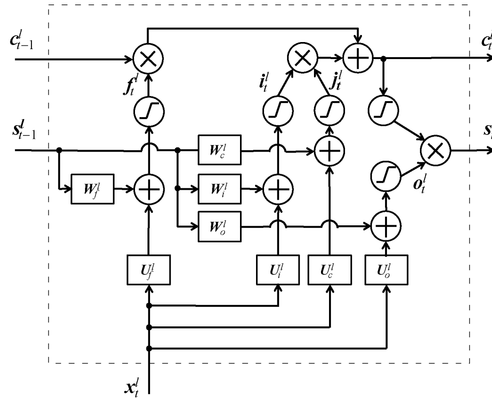


图 11-3　LSTM 单元的内部结构图

与基本 RNN 单元不同，LSTM 的输出包含一个记忆向量（即第 8 章提到的“细胞状态”），表示综合了过去时刻记忆和当前时刻输入等信息后得到的新记忆。LSTM 单元的输入为前一时刻隐藏层的状态向量 $\boldsymbol{s}_{t-1}^{l}$、前一时刻所得到的记忆向量 $\boldsymbol{c}_{t-1}^{l}$，以及当前时刻的输入向量 $\boldsymbol{x}_{t}^{l}$；输出为当前时刻的隐藏层状态向量 $\boldsymbol{s}_{t}^{l}$ 和记忆向量 $\boldsymbol{c}_{t}^{l}$。

LSTM 内部包含输入门（input gate）、忘记门（forget gate）、记忆门（memory gate）和输出门（output gate）。输入门根据 $\boldsymbol{x}_{t}^{l}$ 和 $\boldsymbol{s}_{t-1}^{l}$ 以及矩阵 $\boldsymbol{W}_{i}^{l}$ 和 $\boldsymbol{U}_{i}^{l}$ 中的参数决定 $\boldsymbol{x}_{t}^{l}$ 对记忆向量 $\boldsymbol{c}_{t}^{l}$ 的影响；记忆门根据 $\boldsymbol{x}_{t}^{l}$ 和 $\boldsymbol{s}_{t-1}^{l}$ 以及矩阵 $\boldsymbol{W}_{c}^{l}$ 和 $\boldsymbol{U}_{c}^{l}$ 中的参数决定中间态向量 $\boldsymbol{j}_{t}^{l}$（用于产生新的记忆向量 $\boldsymbol{c}_{t}^{l}$）；遗忘门根据 $\boldsymbol{x}_{t}^{l}$ 和 $\boldsymbol{s}_{t-1}^{l}$ 以及矩阵 $\boldsymbol{W}_{f}^{l}$ 和 $\boldsymbol{U}_{f}^{l}$ 中的参数决定前一时刻的记忆向量 $\boldsymbol{c}_{t-1}^{l}$ 对当前记忆向量 $\boldsymbol{c}_{t}^{l}$ 的影响；输出门根据 $\boldsymbol{x}_{t}^{l}$、$\boldsymbol{s}_{t-1}^{l}$ 和 $\boldsymbol{c}_{t}^{l}$ 以及矩阵 $\boldsymbol{W}_{o}^{l}$ 和 $\boldsymbol{U}_{o}^{l}$ 中的参数决定隐藏层的输出状态向量 $\boldsymbol{s}_{t}^{l}$。在 LSTM 单元中，记忆向量只沿时间序列方向（或水平方向）传递，而隐藏层状态向量则同时沿深度方向和时间序列方向传递。

在 tensorflow/python/ops/rnn_cell_impl.py 文件中，继承于 `RNNCell` 的 `BasicLSTMCell` 类实现了最基本的 LSTM 单元，它们不具备一些 LSTM 的高级特性，如单元裁剪（cell clipping）、投 影层（projection layer）、窥视孔连接（peep-hole connection）等。在其 `call` 方法中，变量 `c` 和 `h`（`new_c` 和 `new_h`）分别表示输入（输出）的记忆向量和隐藏层状态向量，`i`、`j`、`f`、`o` 分别对应图 11-3 中的 $\boldsymbol{f}_{t}^{l}$、$\boldsymbol{i}_{t}^{l}$、$\boldsymbol{j}_{t}^{l}$、$\boldsymbol{o}_{t}^{l}$ 激活前的值。`_linear` 方法用于实现输入门、忘记门、输出门和记忆门所对应的矩阵计算。`j` 和 `new_c` 对应的激活函数是 tanh 或者用户自定义的函数，其余用到的激活函数为

sigmoid。该 `call` 方法有两个输出——`new_h` 和 `new_state`，其中前者用于下一层 RNN 模块的输入，后者表示当前时刻隐藏层状态向量和记忆向量。`BasicLSTMCell` 类采用 `LSTMStateTuple` 数据结构，以 tuple 的形式顺序保存记忆向量和当前时刻的隐藏层状态向量。该 `call` 方法的定义如下：

```
def call(self, inputs, state):
  sigmoid = math_ops.sigmoid
  if self._state_is_tuple:
    c, h = state
  else:
    c, h = array_ops.split(value=state, num_or_size_splits=2, axis=1)
  concat = _linear([inputs, h], 4 * self._num_units, True)
  i, j, f, o = array_ops.split(value=concat, num_or_size_splits=4, axis=1)
  new_c = (c * sigmoid(f + self._forget_bias) + sigmoid(i) * self._activation(j))
  new_h = self._activation(new_c) * sigmoid(o)
  if self._state_is_tuple:
    new_state = LSTMStateTuple(new_c, new_h)
  else:
    new_state = array_ops.concat([new_c, new_h], 1)
  return new_h, new_state
```

11.1.4　GRU单元

GRU（Gated Recurrent Unit）是 RNN 单元的又一个变种，也可以解决长时间序列的记忆问题，常用于机器翻译等任务。与 LSTM 单元不同，GRU 单元的记忆向量并不会显式地在不同时刻之间传递。GRU 单元比 LSTM 单元更为简单。从大量实验结果来看，GRU 单元训练的收敛速度比 LSTM 单元要稍微快一些。

GRU 单元的内部结构如图 11-4 所示。该单元包含三种门：重置门（reset gate）、更新门（update gate）和记忆门（memory gate）。重置门根据 $\boldsymbol{x}_t^l$ 和 $\boldsymbol{s}_{t-1}^l$ 以及矩阵 $\boldsymbol{W}_r^l$ 和 $\boldsymbol{U}_r^l$ 中的参数决定 $\boldsymbol{s}_{t-1}^l$ 对新的记忆向量产生的影响；记忆门根据重置门的输出向量与 $\boldsymbol{x}_t^l$ 以及矩阵 $\boldsymbol{W}^l$ 和 $\boldsymbol{U}^l$ 中的参数共同决定了当前新的记忆向量；更新门根据 $\boldsymbol{x}_t^l$ 和 $\boldsymbol{s}_{t-1}^l$ 以及矩阵 $\boldsymbol{W}_u^l$ 和 $\boldsymbol{U}_u^l$ 中的参数决定 $\boldsymbol{s}_{t-1}^l$ 对 $\boldsymbol{s}_t^l$ 产生的影响。

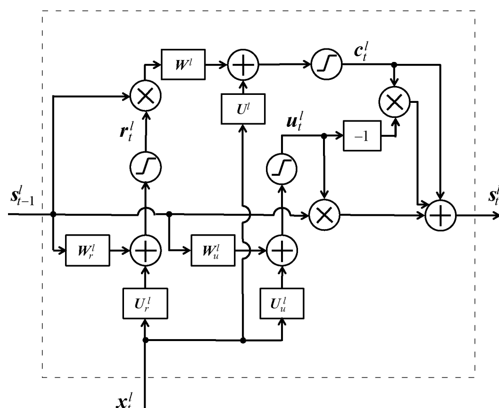


图 11-4　GRU 单元的内部结构图

在 tensorflow/python/ops/rnn_cell_impl.py 文件中，继承于 `RNNCell` 的 `GRUCell` 类实现了 GRU 单元。在其 `call` 方法中，参数 `inputs` 和 `state` 表示输入向量和前一时刻的隐藏层状态向量，变量 `r` 和 `u` 表示重置门和更新门的输出向量，`c` 表示由输入 `inputs`、`r*state` 及其相应的参数矩阵计算得到的记忆向量，`new_h` 表示当前时刻的隐藏层状态向量（同时也表示输出向量）。该方法在计算 `r` 和 `u` 时采用的激活函数是 sigmoid，而计算 `c` 时采用的激活函数一般为 tanh，也可以是用户自定义的激活函数。相关代码如下：

```
def call(self, inputs, state):
  with vs.variable_scope("gates"):
    bias_ones = self._bias_initializer
    if self._bias_initializer is None:
      dtype = [a.dtype for a in [inputs, state]][0]
      bias_ones = init_ops.constant_initializer(1.0, dtype=dtype)
    value = math_ops.sigmoid(
        _linear([inputs, state], 2 * self._num_units, True, bias_ones,
                self._kernel_initializer))
    r, u = array_ops.split(value=value, num_or_size_splits=2, axis=1)
  with vs.variable_scope("candidate"):
    c = self._activation(
        _linear([inputs, r * state], self._num_units, True,
                self._bias_initializer, self._kernel_initializer))
  new_h = u * state + (1 - u) * c
  return new_h, new_h
```

11.1.5　双向RNN单元

在对句子进行语义分析时，当前词的语义要与上下文语义匹配。这里所说的“上下文”不仅包括“上文”，也包括“下文”。但是，图 11-2 所示的基本 RNN 单元仅考虑了从 $t-1$ 时刻到 t 时刻的状态转移，即之前时刻的输入对当前的影响。为了考虑未来时刻的输入对当前的影响，充分利用上下文的语义信息，Mike Schuster 等人发明了双向 RNN 单元。如图 11-5 所示，双向 RNN 单元中包含两个隐藏层状态，它在 RNN 单元的基础上添加了相反方向的隐藏层状态转移。在 t 时刻，第 l 层的正反向的隐藏层状态分别用 $\boldsymbol{s}_t^l$ 和 $\boldsymbol{h}_t^l$ 表示。在正向（从左到右）传播路径上的线性变换矩阵用 $\boldsymbol{W}_f^l$ 和 $\boldsymbol{U}_f^l$ 表示，在反向传播路径上的线性变换矩阵用 $\boldsymbol{W}_b^l$ 和 $\boldsymbol{U}_b^l$ 表示。双向 RNN 单元本质上可以由两个独立的基本 RNN 单元组合而成，在构造双向 RNN 模型时，我们可以直接使用两个基本 RNN 单元，因此 TensorFlow 没有提供继承于 `RNNCell` 类的双向 RNN 单元子类。我们也可以借鉴双向 RNN 单元的思想构建双向 LSTM、双向 GRU 等单元。

从双向 RNN 单元的原理来看，它适合处理同时具有前后依赖关系的序列型输入数据。除了前面提到的句子语义分析之外，也会有其他场景需要用到双向 RNN 单元。例如，在光学字符识别（OCR）中，输入序列中每一时刻对应一字母。对于任何一个单词中的某一个字母，其出现的概率与其之前和之后的字母都有一定关系。因此，当我们观察到具体应用中的这种特征之后，就可以选择双向 RNN 单元构建模型，以提升模型准确率。

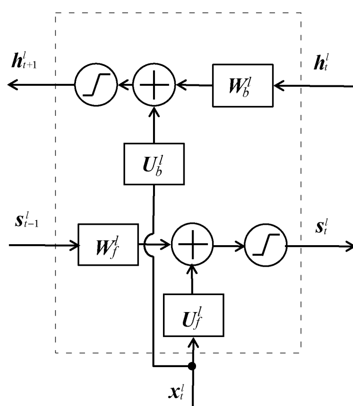


图 11-5　双向 RNN 单元的内部结构图

11.1.6　带有其他特性的RNN单元

基本 RNN 单元、LSTM 单元、GRU 单元等还可以结合其他特性，从而形成新的 RNN 单元变种。tensorflow/python/ops/rnn_cell_impl.py 文件定义的 `DropoutWrapper`、`ResidualWrapper`、`DeviceWrapper`、`MultiRNNCell` 等包装器或工具类都继承自 `RNNCell` 类，它们分别提供不同的特性。其中，`DeviceWrapper` 比较简单，主要是使某个具体的 RNN 单元运行在指定的设备上，这里就不介绍了。下面简要其他几个类。

1. `DropoutWrapper`

该类提供 Dropout 特性，可以在一定程度上避免模型太大造成的过拟合问题。在训练过程的每次迭代中，输入向量、隐藏层状态向量和输出向量对应的神经元分别经过 Dropout 层以屏蔽向量中的部分值，使之不参与后续计算。在 `DropoutWrapper` 对象初始化时，需要指定 `input_keep_prob`、`output_keep_prob` 和 `state_keep_prob` 参数。这三个参数分别给出输入向量、隐藏层状态向量和输出向量的神经元参与到后续计算的比例。相应代码如下：

```
def __call__(self, inputs, state, scope=None):
  def _should_dropout(p):
    return (not isinstance(p, float)) or p < 1
  if _should_dropout(self._input_keep_prob):
    inputs = self._dropout(inputs, "input", self._recurrent_input_noise,
self._input_keep_prob)
  output, new_state = self._cell(inputs, state, scope)
  if _should_dropout(self._state_keep_prob):
    new_state = self._dropout(new_state, "state", self._recurrent_state_noise,
self._state_keep_prob)
  if _should_dropout(self._output_keep_prob):
    output = self._dropout(output, "output", self._recurrent_output_noise,
self._output_keep_prob)
  return output, new_state
```

2. ResidualWrapper

Residual 特性是指输入可以直接与其经非线性变换后的输出组合在一起的特性，这使得后向梯度的传播更容易。9.1 节介绍的 ResNet module 就具有这种 Residual 特性。ResidualWrapper 类提供 Residual 特性，其关键代码如下所示：

```
def __call__(self, inputs, state, scope=None):
  outputs, new_state = self._cell(inputs, state, scope=scope)
  nest.assert_same_structure(inputs, outputs)
  def assert_shape_match(inp, out):
    inp.get_shape().assert_is_compatible_with(out.get_shape())
  nest.map_structure(assert_shape_match, inputs, outputs)
  res_outputs = nest.map_structure(lambda inp, out: inp + out, inputs, outputs)
  return (res_outputs, new_state)
```

该类的 __call__ 方法首先得到输出向量 outputs 和当前时刻的隐藏层状态向量 new_state，然后在保证输入向量 inputs 和输出向量 outputs 维度一致的条件下，将输入向量和输出向量叠加后作为新的输出向量 res_outputs。

具有这种特性的 RNN 模型可用于对输入和输出之间的残差进行建模。具有 Residual 特性的 RNN 模型有利于解决梯度消失现象，其模型深度可以比其他 RNN 模型更深。

3. MultiRNNCell

当训练数据量增大时，增加模型深度往往能够获得更好的精度。很多自然语言处理任务（如机器翻译）都需要用到多层 RNN 模型。MultiRNNCell 用于将多个 RNN 堆叠起来形成多层 RNN，并将它们合并为一个单元。该类的 call 方法的代码如下：

```
def call(self, inputs, state):
  cur_state_pos = 0
  cur_inp = inputs
  new_states = []
  for i, cell in enumerate(self._cells):
    with vs.variable_scope("cell_%d" % i):
      if self._state_is_tuple:
        if not nest.is_sequence(state):
          raise ValueError(
              "Expected state to be a tuple of length %d, but received: %s" %
              (len(self.state_size), state))
        cur_state = state[i]
      else:
        cur_state = array_ops.slice(state, [0, cur_state_pos], [-1, cell.state_size])
        cur_state_pos += cell.state_size
      cur_inp, new_state = cell(cur_inp, cur_state)
      new_states.append(new_state)
  new_states = (tuple(new_states) if self._state_is_tuple else
                array_ops.concat(new_states, 1))
  return cur_inp, new_states
```

该方法通过遍历多个 cell，将前一个 cell 的输出向量作为下一个 cell 的输入向量，并将其中的隐藏层状态向量保存在 new_states 中。返回的向量为最后一个 cell 的输出向量和所有

cell 的隐藏层状态向量。

此外，tensorflow/contrib/rnn/python/ops/core_rnn_cell.py 文件提供了 EmbeddingWrapper、InputProjectionWrapper 和 OutputProjectionWrapper 三种类。它们也都继承自 RNNCell，分别提供带有输入嵌入、输入映射和输出映射特性的 RNN 单元。

11.2 RNN 模型

在了解了 RNN 模型的基本组成部分之后，我们有必要通过实例来加深对这类模型的理解。本节介绍两个经典的 RNN 模型——PTB-LSTM 语言模型和 Seq2Seq 模型。PTB-LSTM 作为一种典型的语言模型，能够预测一个句子产生的概率，常用于语音识别、语句补全等自然语言处理任务。Seq2Seq 模型可以用于建立两个时间序列数据之间的关系，主要应用场景包括智能问答、机器翻译、文本摘要等。分析这两个模型的实现原理，有助于读者体会 RNN 在利用输入数据的时间序列特性的设计精髓。

11.2.1 PTB-LSTM语言模型

8.2.2 节已经简单描述了语言模型的概念。我们可以通过语言模型得到一个句子出现的概率并评价句子的合法性。PTB-LSTM 语言模型是基于 PTB（Penn Treebank）数据集和 LSTM 模型构建的语言模型。在介绍模型的实现细节之前，我们先了解一下 PTB 数据集。PTB 数据集是美国宾西法尼亚大学 LINC（Language Information and Computation）实验室搜集和标注（包括词性标注和句法分析）的一套文本数据集。PTB 数据集的主要语料来源是 1989 年《华尔街日报》中的 2499 篇文章（约 100 万词汇），该数据集经常用作训练语言模型的语料库。

TensorFlow 社区 models 项目的 tutorials/rnn/ptb 目录提供了 PTB-LSTM 语言模型的训练程序。该目录中有三个 Python 文件：reader.py、reader_test.py 和 ptb_model_lm.py。其中，reader.py 用于读取 PTB 数据集的句子和单词并进行相应的预处理，reader_test.py 用于测试 reader.py 中的几个主要方法，ptb_model_lm.py 用于实现 PTB-LSTM 语言模型及其训练步骤。

reader.py 提供的方法包括用于内部逻辑实现的 _read_words、_build_vocab 和 _file_to_word_ids，以及作为对外功能接口的 ptb_raw_data 和 ptb_producer。前三个内部逻辑实现方法主要用于读取数据集并标识单词。其中，_read_words 方法用于读取指定数据集中的每个单词；_build_vocab 方法用于得到字典 word_to_id，该字典保存了指定数据集中每个单词及其所对应的 ID；_file_to_word_ids 方法用于得到指定数据集中每个单词的 ID。

在 reader.py 的对外方法中，ptb_raw_data 方法封装了上述内部方法，用于得到训练、验证和测试所需的数据集各自包括的单词 ID（分别为 train_data、valid_data、test_data）以及训练集对应的字典（word_to_id）长度。其代码为：

```
def ptb_raw_data(data_path=None):
  train_path = os.path.join(data_path, "ptb.train.txt") # 训练数据集
```

```
    valid_path = os.path.join(data_path, "ptb.valid.txt") # 验证数据集
    test_path = os.path.join(data_path, "ptb.test.txt") # 测试数据集
    word_to_id = _build_vocab(train_path)
    train_data = _file_to_word_ids(train_path, word_to_id)
    valid_data = _file_to_word_ids(valid_path, word_to_id)
    test_data = _file_to_word_ids(test_path, word_to_id)
    vocabulary = len(word_to_id)
    return train_data, valid_data, test_data, vocabulary
```

ptb_producer 方法用于给 PTB-LSTM 语言模型提供张量类型的输入数据对 $(\boldsymbol{x},\boldsymbol{y})$。$\boldsymbol{x}$ 和 $\boldsymbol{y}$ 均为形如[batch_size,num_steps]的二维张量，张量元素是数据集中的单词 ID。假设 $\boldsymbol{x}$ 中每个单词 ID 所对应的单词为 $\boldsymbol{x}(i,j)$，那么 $\boldsymbol{y}$ 中相应位置的 ID 对应的则是原始句子中 $\boldsymbol{x}(i,j)$的下一个单词。我们可以将 $\boldsymbol{x}$ 作为语言模型的输入数据，$\boldsymbol{y}$ 作为输入数据的“标签”。通过不断迭代训练，该模型将具备根据已有单词预测下一个单词的能力。ptb_producer 方法的具体代码为：

```
def ptb_producer(raw_data, batch_size, num_steps, name=None)
  with tf.name_scope(name, "PTBProducer", [raw_data, batch_size, num_steps]):
    # 将 ptb_raw_data 方法获得的某一数据集（train_data、valid_data 和 test_data 之一）
    # 中的所有单词对应的 ID 转化为 tf.int32 类型的张量，并仍保存为 raw_data
    raw_data = tf.convert_to_tensor(raw_data, name="raw_data", dtype=tf.int32)
    data_len = tf.size(raw_data)
    batch_len = data_len // batch_size
    # 将 raw_data 变为二维张量 data，第一个维度 batch_size 表示批的大小，
    # 第二个维度 batch_len 表示该数据集包含的批的长度
    data = tf.reshape(raw_data[0 : batch_size * batch_len], [batch_size, batch_len])
    # num_steps 表示时间序列方向上 LSTM 单元的个数，即 LSTM 模型的输入的单词数量；
    # epoch_size 表示数据集包含的批的数量
    epoch_size = (batch_len - 1) // num_steps
    assertion = tf.assert_positive(epoch_size, message="epoch_size == 0, decrease batch_size
        or num_steps")
    with tf.control_dependencies([assertion]):
      epoch_size = tf.identity(epoch_size, name="epoch_size")
    # 顺序产生从 0 到(epoch_size-1)的序号 i
    i = tf.train.range_input_producer(epoch_size, shuffle=False).dequeue()
    # 张量 x 和 y 的形状均为[batch_size, num_steps]，通过 i 的变化可以遍历数据集中的全部单词
    x = tf.strided_slice(data, [0, i * num_steps], [batch_size, (i + 1) * num_steps])
    x.set_shape([batch_size, num_steps])
    y = tf.strided_slice(data, [0, i * num_steps + 1], [batch_size, (i + 1) * num_steps + 1])
    y.set_shape([batch_size, num_steps])
    return x, y
```

ptb_model_lm.py 定义了用于表示输入数据的 PTBInput 类、用于表示语言模型的 PTBModel 类、四个参数配置类——SmallConfig、MediumConfig、LargeConfig 和 TestConfig，以及用于训练或推理的 run_epoch 等方法。

- PTBInput 类是输入数据类，其对象初始化代码如下所示：

```
class PTBInput(object):
  def __init__(self, config, data, name=None):
    self.batch_size = batch_size = config.batch_size
    self.num_steps = num_steps = config.num_steps
    self.epoch_size = ((len(data) // batch_size) - 1) // num_step
```

```
self.input_data, self.targets = reader.ptb_producer(data, batch_size, num_steps,
    name=name)
```

它通过调用 reader.py 的 `ptb_producer` 方法得到输入数据 `input_data` 和目标数据 `targets`。

我们期望基于 LSTM 训练出来的语言模型可以根据之前输入的单词自动预测出下一个单词，从而可以用于生成句子或者判别句子产生的概率。该语言模型的示意图如图 11-6 所示。

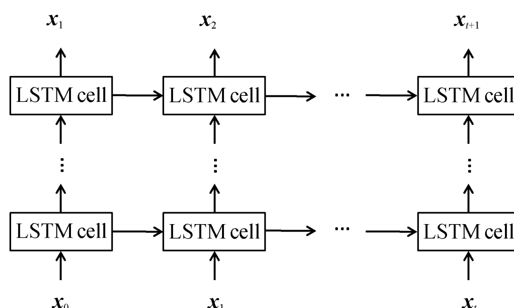


图 11-6 基于 LSTM 的语言模型示意图

- `PTBModel` 类主要用于根据用户的配置信息和输入数据等构建基于 LSTM 的语言模型。该类的初始化函数完成超参设置和模型建立。其他函数（如 `_build_rnn_graph`、`_build_rnn_graph_cudnn`、`build_rnn_graph_lstm`、`_get_lstm_cell`）通过调用 TensorFlow 中的 LSTM 接口，辅助语言模型的建立。此外，`PTBModel` 类还提供了 `import_ops` 和 `export_ops` 函数，方便用户从搜集器（`collection`）中导入和导出模型会用到的操作（如 `_train_op`）。

`PTBModel` 类的初始化函数如下：

```
def __init__(self, is_training, config, input_):
  self._is_training = is_training # 判断该模型是否需要被训练
  self._input = input_ # input_是一个 PTBInput 对象，用于表示模型的输入数据
  self._rnn_params = None # 模型参数
  self._cell = None # 组成模型的 RNN 单元
  self.batch_size = input_.batch_size # 输入数据的批大小
  self.num_steps = input_.num_steps # 输入数据的长度（以单词为单位）
  size = config.hidden_size # 模型第一层 RNN 单元中隐藏层神经元的数量，即每个输入单词词向量的长度
  vocab_size = config.vocab_size # 数据集中单词的数量
  # 在 CPU 上完成词向量查询
  with tf.device("/cpu:0"):
    # embedding 表示词向量矩阵。该矩阵的行数和列数分别为 vocab_size 和 size，
    # 矩阵的每一行表示一个单词的词向量
    embedding = tf.get_variable("embedding", [vocab_size, size], dtype=data_type())
    # 通过 input_.input_data 中的单词 ID，在词向量矩阵 embedding 中查询单词的词向量
    inputs = tf.nn.embedding_lookup(embedding, input_.input_data)
  # 如果当前模型处于训练态，并且 config 对象中设置了 Dropout 层的参数 keep_prob(该参数值小于 1),
  # 则输入数据（已转换为词向量）在输入到模型之前需要先经过 Dropout 层。
  # 采用 Dropout 层的好处在于可以在一定程度上减缓模型过拟合的风险
  if is_training and config.keep_prob < 1:
```

11

```
  inputs = tf.nn.dropout(inputs, config.keep_prob)
# 通过内部调用 cuDNN 的 LSTM 接口或者 TensorFlow 的 LSTM 接口（具体选择由 config 中的 rnn_mode 决定）
# 建立 LSTM 模型
output, state = self._build_rnn_graph(inputs, config, is_training)
# 输入数据经过 LSTM 模型处理后进入输出层。在输出层中，
# 矩阵 softmax_w 的转置与每个 LSTM 单元输出向量相乘后，再与 softmax_b 相加，得到 logits。
# logits 用于表示输出单词在训练集词库中的索引
softmax_w = tf.get_variable("softmax_w", [size, vocab_size], dtype=data_type())
softmax_b = tf.get_variable("softmax_b", [vocab_size], dtype=data_type())
logits = tf.nn.xw_plus_b(output, softmax_w, softmax_b)
# 为了满足 tf.contrib.seq2seq.sequence_loss 接口对 logits 形状的要求，将 logits 的维度变为
# [batch_size, num_steps, vocab_size]
logits = tf.reshape(logits, [self.batch_size, self.num_steps, vocab_size])
# 通过比较 logits 和 input_.targets，得到当前输入数据在 batch_size 维度上损失函数的平均值 loss
loss = tf.contrib.seq2seq.sequence_loss(
    logits,
    input_.targets,
    tf.ones([self.batch_size, self.num_steps], dtype=data_type()),
    average_across_timesteps=False,
    average_across_batch=True)
# 将张量 loss 中的所有值归约到一个值，并保存在 PTBModel 对象的成员变量 _cost 中
self._cost = tf.reduce_sum(loss)
self._final_state = state
if not is_training:
  return
self._lr = tf.Variable(0.0, trainable=False)
tvars = tf.trainable_variables()
# 为了防止梯度爆炸，需要根据 config.max_grad_norm 设定的上限对梯度进行裁剪
grads, _ = tf.clip_by_global_norm(tf.gradients(self._cost, tvars), config.max_grad_norm)
optimizer = tf.train.GradientDescentOptimizer(self._lr)
self._train_op = optimizer.apply_gradients(zip(grads, tvars),
    global_step=tf.contrib.framework.get_or_create_global_step())
# 允许学习速率在训练过程中被更新
self._new_lr = tf.placeholder(tf.float32, shape=[], name="new_learning_rate")
self._lr_update = tf.assign(self._lr, self._new_lr)
```

- ptb_model_lm.py 提供的四个预置的配置类（`SmallConfig`、`MediumConfig`、`LargeConfig` 和 `TestConfig`）比较简单。它们没有成员函数，只是简单地罗列出配置参数的键值对。这些参数主要涉及 `PTBModel` 类所用到的一些超参，包括跟模型大小相关的超参（如 `num_steps` 和 `num_layers` 等），以及跟训练相关的超参（如 `batch_size`、`learning_rate`、`lr_decay` 等）。这四个配置类中所有参数所对应的键都一样，只是值不一样。用户可以在这四个配置类中添加新的配置参数使得模型训练或推理更灵活。
- 在训练 PTB-LSTM 语言模型时，我们通常要遍历多次训练数据集。遍历一次全量数据称为完成一个 epoch。ptb_model_lm.py 提供的 `run_epoch` 方法用于迭代运行一个已创建的 `PTBModel` 模型对象，并在每次迭代完成后获得该模型对象的损失函数值。直至完成一个 epoch 的预设迭代次数之后，输出该模型对象的困惑度。困惑度是描述语言模型准确性的关键指标，与损失函数值正相关。困惑度越低，语言模型越准确。

综上，我们理解了 tutorials/rnn/ptb 目录提供的 PTB-LSTM 语言模型训练框架的基本原理。

用户可以模仿 PTB 数据的读取和预处理过程以及 PTBModel 的定义，在 `reader` 模块和 `ptb_model_lm` 中添加自定义的数据集和语言模型，从而复用这套框架训练一个新的语言模型。

11.2.2 Seq2Seq模型

Seq2Seq 模型是用来学习一对时间序列数据之间映射关系的模型，它能够应用于智能问答、文本摘要、机器翻译、语音识别等领域。该模型可以基于基本 RNN 单元或其变种来构建，本节主要介绍基于 LSTM 单元的 Seq2Seq 模型。如图 11-7 所示，该模型主要包括两个部分——编码器和解码器，二者均为一层或多层的 LSTM 模型。解码器中第一个输入 GO 表示输入句子的开始，最后一个输出 EOS（End Of Sentence）表示输出句子的结束。

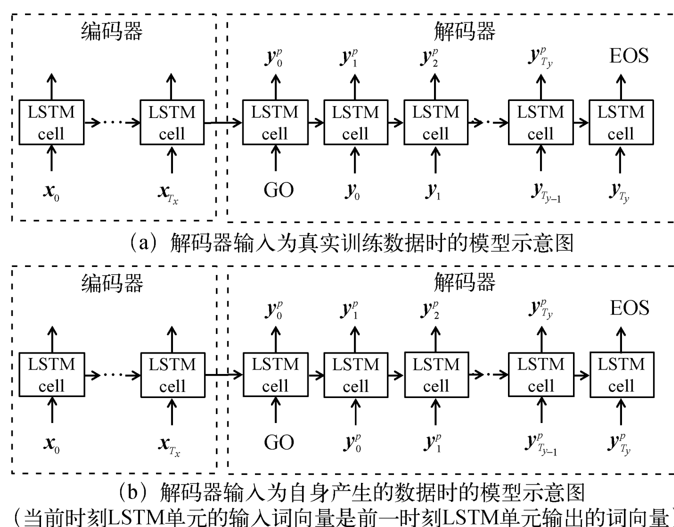


图 11-7 基于 LSTM 单元的 Seq2Seq 模型示意图

11

下面我们以英法句子翻译为例，介绍如何训练最基础的 Seq2Seq 模型。在训练数据中，假设 $\boldsymbol{x}_0,\cdots,\boldsymbol{x}_{T_x}$ 为某个英语句子中每个单词的词向量，$\boldsymbol{y}_0,\cdots,\boldsymbol{y}_{T_y}$ 为语义相同的法语句子中每个单词的词向量。编码器通过一层或者多层 LSTM 将输入的英语句子的语义用上下文向量 $\boldsymbol{c}=g(\boldsymbol{h}_0,\boldsymbol{h}_1,\cdots,\boldsymbol{h}_{T_x})$ 表示，其中 $g()$为自定义的非线性函数，$\boldsymbol{h}_0,\boldsymbol{h}_1,\cdots,\boldsymbol{h}_{T_x}$ 为编码器中每个 LSTM 单元的隐藏层状态。为简单起见，在有些 Seq2Seq 模型的解码器中，上下文向量 $\boldsymbol{c}$ 直接等于 $\boldsymbol{h}_{T_x}$。在解码器中，t 时刻的输出向量取决于 t 时刻的输入向量、$t-1$ 时刻隐藏层状态向量以及编码器中的上下文向量 $\boldsymbol{c}$。如果不考虑编码器输出的上下文向量 $\boldsymbol{c}$ 的影响，解码器就可以看作前面提到的语言模型。一般而言，当 Seq2Seq 模型处于训练态时，解码器在 t 时刻的输入向量为训练数据中的标签数据 $\boldsymbol{y}_t$，输出向量为解码器在当前时刻的预测向量 $\boldsymbol{y}_t^p$，如图 11-7a 所示；当该模型处于推理态时，解码器在 t 时刻的输入向量为其在 $t-1$ 时刻的输出向量 $\boldsymbol{y}_{t-1}^p$，输出向量为当前时刻的预测向量 $\boldsymbol{y}_t^p$，如图 11-7b 所示。这样依次循环下去，最终 Seq2Seq 模型输出了原英文句子对应的法语句子。因此，一般情

况下，Seq2Seq 模型的解码器在训练态和推理态时采用了两种截然不同的输入方式。不过，也存在一些情况，处于训练态的 Seq2Seq 模型可以融合这两种输入方式。比如 Google 的研究员 Samy Bengio 等人于 2015 年提出了一种方法，该方法以一定的概率决定 t 时刻解码器的输入是来自前一时刻解码器的输出，还是来自真实的训练数据。这种方法的好处在于有可能降低解码器在训练态和推理态之间的差距，从而提升模型的泛化性。

在上述模型中，解码器计算每个时刻的输出单词时，用到的上下文向量 $\boldsymbol{c}$ 都是一样的。为了更好地利用编码器中的输入信息，Yoshua Bengio 等人提出了具有注意力（attention）机制的 Seq2Seq 模型。在这种模型的解码器中，i 时刻的输出单词所依赖的上下文向量为 $\boldsymbol{c}_i=\sum_{j=0}^{T_x}\boldsymbol{h}_j\exp(\boldsymbol{q}_{ij})/\sum_{k=0}^{T_x}\exp(\boldsymbol{q}_{ik})$。其中，$\boldsymbol{h}_j$ 为编码器中 j 时刻隐藏层状态向量，$\boldsymbol{q}_{ij}=a(\boldsymbol{s}_{i-1},\boldsymbol{h}_j)$ 用于表示 $\boldsymbol{h}_j$ 和解码器中 $t-1$ 时刻的隐藏层状态向量 $\boldsymbol{s}_{i-1}$ 之间的吻合度（$a()$ 是用前馈神经网络表示的非线性函数）。这种注意力机制使得每个时刻输出的单词所依赖的上下文向量不同。每个时刻的输出可以自适应地选择中间隐藏层状态的组合，从而提升模型精度。其实，注意力机制是一种通用的机制，它最早被应用于计算机视觉中的一些任务（如显著性目标检测）。除了 Seq2Seq 模型，注意力机制还可以应用于其他 RNN 模型。

在了解 Seq2Seq 模型的基本原理之后，我们结合代码讲解如何使用 TensorFlow 实现该模型。TensorFlow 社区 models 项目的 tutorials/rnn/translate 目录包含一套英法翻译模型的简单实现，其主体代码位于 3 个文件：data_utils.py、seq2seq_model.py 和 translate.py。其中，data_utils.py 定义了一些数据准备和预处理函数，translate.py 提供了基于英法翻译数据集的训练和预测功能，seq2seq_model.py 定义了 Seq2Seq 模型。下面重点介绍 seq2seq_model.py。

seq2seq_model.py 只定义了一个类，即 `Seq2SeqModel`。该类除了包含用于构建模型的初始化方法之外，还提供 `get_batch` 和 `step` 方法，它们分别用于获取单次迭代的批训练数据和执行单次迭代的模型计算。`Seq2SeqModel` 类的初始化方法用到的基础模型主要来自于 tensorflow/contrib/legacy_seq2seq/python/ops/seq2seq.py 文件。seq2seq.py 定义了基本的 Seq2Seq 模型（`basic_rnn_seq2seq`）及其变种。这些变种包括 `tied_rnn_seq2seq`、`embedding_rnn_seq2seq`、`embedding_tied_rnn_seq2seq`、`embedding_attention_seq2seq` 等模型。它们主要是在 `basic_rnn_seq2seq` 基础上集成 tied、embedding 和 attention 中的一个或多个特性。其中，tied 特性是指编码器和解码器中的 LSTM 参数值可以共享，以减少模型参数的规模；embedding 特性是指编码器和解码器的输入张量（用于标识单词的离散值）在进入 LSTM 单元之前都需要先经过 embedding 层；attention 特性是指利用前面提到的注意力机制改进模型。

下面我们以 basic_rnn_seq2seq 为例，介绍 Seq2Seq 模型的代码实现。该方法的定义如下：

```
def basic_rnn_seq2seq(encoder_inputs, decoder_inputs, cell, dtype=dtypes.float32, scope=None):
  with variable_scope.variable_scope(scope or "basic_rnn_seq2seq"):
    # enc_cell 可以为 LSTM、GRU 或者其他 RNN 单元
    enc_cell = copy.deepcopy(cell)
    _, enc_state = rnn.static_rnn(enc_cell, encoder_inputs, dtype=dtype)
    return rnn_decoder(decoder_inputs, enc_state, cell)
```

在这个方法内部，我们调用了 tensorflow.python.ops.rnn 模块的 static_rnn 方法。该方法用于构建编码器（即一个静态的 RNN 模型）。它的输入参数包括 enc_cell、encoder_inputs 和 dtype，它们的含义如下。

- ❑ enc_cell 表示构建编码器所使用的 RNN 单元，本例中为 LSTM 单元。
- ❑ encoder_inputs 表示编码器的输入，它是由一系列形如 [batch_size, input_size] 的二维张量组成的列表，列表长度 num_steps 等于编码器沿时间序列方向展开后的长度，batch_size 表示当前训练数据的批大小，input_size 表示编码器中每个 LSTM 单元允许的输入词向量的长度。
- ❑ dtype 表示输入输出状态的数据类型。

static_rnn 的输出包括编码器在各个时刻的输出向量（记为 enc_outputs）和最后一个时刻的隐藏层状态向量（即上下文向量 enc_state）。因为 enc_outputs 对于解码器没用，所以上述代码只关心当前 batch 内每个输入的句子对应的上下文向量 enc_state，并将其输入到解码器中。

在 basic_rnn_seq2seq 方法末尾，我们调用的 rnn_decoder 方法用于构建解码器。rnn_decoder 方法的输入参数包括 decoder_inputs、initial_state 和 cell 等，其中前两个参数的含义如下。

- ❑ decoder_inputs 表示解码器的输入。类似于编码器的输入 encoder_inputs，它也是由一系列二维张量组成的列表。
- ❑ initial_state 表示编码器给解码器输入的上下文向量，其形状为 [batch_size, cell.state_size]，其中 cell.state_size 表示解码器中每个 LSTM 单元的隐藏层向量长度。

在 basic_rnn_seq2seq 方法中，当调用 rnn_decoder 方法时，initial_state 被赋值为编码器输出的上下文向量 enc_state。rnn_decoder 的输出包括解码器在各个时刻的输出向量和最后一个时刻的隐藏层状态向量。rnn_decoder 方法的代码如下：

```
def rnn_decoder(decoder_inputs, initial_state, cell, loop_function=None, scope=None):
  with variable_scope.variable_scope(scope or "rnn_decoder"):
    state = initial_state # state 表示解码器的初始隐藏层状态，一般为编码器输出的隐藏层状态
    outputs = [] # outputs 表示解码器的输出，维度与 decoder_inputs 一致
    prev = None # prev 表示解码器中前一时刻 LSTM 单元的输出
    for i, inp in enumerate(decoder_inputs):
      # 如果采用 loop_function，那么解码器中前一时刻的输出会作为当前时刻的输入。
      # 这种情况常见于 Seq2Seq 模型的推理态（或者某些训练态，如前面提到的 Samy Bengio 等人的方法）
      if loop_function is not None and prev is not None:
        with variable_scope.variable_scope("loop_function", reuse=True):
          inp = loop_function(prev, i)
      if i > 0:
        # 不同时刻对应的 LSTM 单元内的参数是共享的
        variable_scope.get_variable_scope().reuse_variables()
      output, state = cell(inp, state)
      outputs.append(output)
      if loop_function is not None:
        prev = output
  return outputs, state
```

seq2seq.py 文件中的 `model_with_buckets` 方法用于创建具有装桶（bucketing）特性的 Seq2Seq 模型。装桶是一种以句子长度为依据的模型训练优化机制，设计这一机制的动机是为了降低对长短不一的句子进行补零操作带来的开销。对句子进行补零原本是为了充分利用 GPU 等设备的并行计算能力，但过多的补零操作反而会增加无谓的计算。装桶技术将长度接近的多个句子分配到同一个桶中，从而减少补零操作。`model_with_buckets` 方法的 `buckets` 参数表示每个桶中输入句子和输出句子的长度对列表。`Seq2SeqModel` 类的初始化方法调用了 `model_with_buckets` 方法，用于为每一个桶建立一个 Seq2Seq 模型（这些模型互相共享模型参数），以提升训练效率。

此外，需要注意的是，`Seq2SeqModel` 类在构建损失函数时用到了 `tf.nn.sampled_softmax_loss`。该损失函数仅在采样后的目标类别（即单词种类）上计算损失值。对于训练态，当语料库比较大、单词类别很多时，这种基于采样的损失函数计算方法可以减少计算量，从而加速训练。但对于推理态，我们希望能够有绝对准确的损失值，因此不需要调用采样过程。

通过对 PTB-LSTM 和 Seq2Seq 模型的学习，我们可以看出，RNN 模型的层数虽然普遍不如 CNN 模型那样很深，但是它在时间序列方向上增加了一个维度，这使得模型的训练和推理计算都会变得更加复杂，并且需要使用很多新技巧。例如，为了提升模型的精度，需要引入注意力机制；为了提升训练效率，需要考虑装桶和采样技术。然而，新增加的维度使得 RNN 模型在时间序列方向上具有记忆能力，因此 RNN 模型更适合于解决具有时间序列特性的建模问题。其实，我们生活中很多问题都具有时间序列特性，比如语音、文本、多轮对话、问答、视频等。如果我们能够利用 RNN 模型解决好这些问题，那么人工智能的更多应用将被催熟和落地。

11.3　小结

RNN 模型是一种隐藏层的输出不仅连接到下一层而且还连接到自身的模型。不同于 CNN 这类前馈型神经网络模型，RNN 的输出和模型之间具有反馈关系，因而能够处理带有时间序列特性的输入数据。RNN 模型由基本 RNN 单元或其变种组成，一个多层的 RNN 模型是由沿水平和垂直方向分布的多个 RNN 单元相互连接而成。LSTM、GRU 等变种单元在基本 RNN 单元上做了改进，它们主要解决了输入长度增长后出现的长时间依赖问题。TensorFlow 为 RNN 模型构建提供了若干接口，我们可以利用它们构建语言模型或 Seq2Seq 模型，以便服务于机器翻译、语音识别等场景。

第四部分

核心揭秘篇

第 12 章

TensorFlow 运行时核心设计与实现

学习了 TensorFlow 的应用开发接口、上层工作流程，以及经典算法模型的数学原理之后，我们已经了解了如何利用 TensorFlow 开发自己的机器学习或深度学习应用程序。如果想进一步理解 TensorFlow 本身的工作机制，那么就有必要深入运行时库内部，探究其设计原理。TensorFlow 的运行时核心主要使用 C++ 语言开发，以动态链接库形态存在。它为上层的 Python 等多语言 API 提供具体的功能实现。本章首先对 TensorFlow 运行时框架进行整体性介绍，给出功能模块的组织结构。继而介绍关键数据结构、公共基础机制、外部环境接口等全局共性设计。后续章节将展开说明重要的功能模块。

12.1 运行时框架概述

本书 1.3 节展示的 TensorFlow 组件结构示意图已经满足了普通用户认识软件基本框架的需求。然而，对于高级用户和二次开发者，有必要以更细致的眼光审视运行时核心库的内部组件，这样才能为理解 TensorFlow 工作原理勾画出清晰的思路。TensorFlow 运行时核心库的内部结构如图 12-1 所示。它的设计遵循层次化、模块化原则，每个层次的模块调用下层模块实现的功能，并为上层模块提供接口。主要对照源代码包，每个层次一般对应到 tensorflow/core 目录的一至两个子目录。

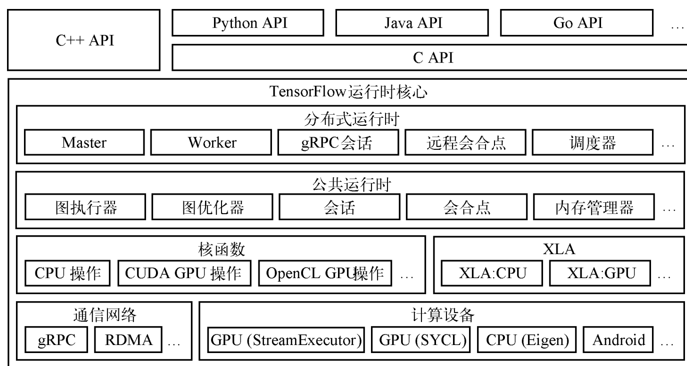


图 12-1　TensorFlow 运行时库的主要模块结构

运行时库的关键数据结构与主要业务逻辑位于公共运行时（common runtime）层次。该层次可认为是 TensorFlow 运行时的中心组件，其代码保存在 common_runtime 子目录。公共运行时围绕数据流图这一基本运行时抽象，实现了图的优化器、执行器等模块。针对计算过程的生命周期管理需求，提供会话抽象及其执行模块。针对进程内设备间数据交换需求，设计了会合点机制。此外，这一层还包括内存管理器等公共组件。理解 TensorFlow 单机运行模式的主体流程，以及分布式运行模式中的本地执行部分，均需要着重分析这一层次。

分布式运行时（distributed runtime）层次服务于 TensorFlow 的分布式运行模式，其代码位于 distributed_runtime 子目录。分布式运行模式以 Master 和 Worker 抽象作为会话管理和数据流图执行的主体，基于 gRPC 远程过程调用库设计了分布式会话逻辑，并提供了跨设备执行数据流图的调度器。同时，该层次在本地会合点基础上封装了远程会合点机制，作为跨进程数据交换的缓冲。理解 TensorFlow 应用中常见的 PS-worker 工作模式及其通信逻辑，需要着重关注这一层次。

核函数层次提供数据流图上操作节点的算法逻辑。数据流图操作的接口定义和注册位于 ops 子目录，核函数针对不同设备的具体实现位于 kernels 子目录。TensorFlow 提供的数据流图操作种类众多，针对不同设备的优化实现各异，因此核函数层次的文件数量相对庞大。对于有兴趣了解算法核函数实现细节或进行细粒度优化的读者，有必要阅读这一层次的源代码。

与核函数功能并列的 XLA（Accelerated Linear Algebra，加速线性代数）模块是一套可选的线性代数计算优化机制，它通过引入中间表示层和专用编译器，以运行时 JIT（just-in-time）或运行前 AOT（ahead-of-time）方式动态生成数据流图算法的可执行代码，取代细粒度的核函数组合，达到提升执行速度、优化内存占用的目标。XLA 为 TensorFlow 引入了编译器领域前后端分离的设计思想，有助于增强对异构加速器设备的可移植性。XLA 相关代码位于 tensorflow/compiler 目录。

运行时库的底层逻辑是通信网络与计算设备接入模块。在通信网络方面，TensorFlow 开源版本通过 gRPC 库访问以太网等 TCP/IP 网络。Google 公司内部版本对 RDMA（远程直接内存访问）通信协议的支持尚未在开源版本中提供，但 tensorflow/contrib/verbs 目录提供了 Yahoo! 公司贡献的另一套 RDMA 通信组件。在计算设备方面，TensorFlow 通过 Eigen、StreamExecutor 等库接入 CPU、GPU 等异构设备。Eigen 库是 TensorFlow 编译安装时的外部依赖项，StreamExecutor 库则直接集成在 TensorFlow 的源代码包中。另外，tensorflow/contrib/android 目录提供对 Android 终端设备的支持组件。

在运行时库的核心模块之上，TensorFlow 封装了多种语言 API，它们的源代码一般位于 tensorflow 目录下以语言名称命名的子目录中。其中 C 语言 API 作为中介层，是 Python、Java、Go 等语言 API 封装实现的基础。需要注意的是，cc 子目录下的 C++ API 同其他语言 API 一样，属于应用层开发接口，不要将其与 core 子目录下的核心层实现代码混淆。

12.2　关键数据结构

为了理解 TensorFlow 运行时框架的设计与实现，进而分析具体组件的功能原理，有必要认识框架内部的关键数据结构。这些数据结构，有些与 Python 接口层的数据结构相对应，是其底

层 C++ 实现；有些则不对应用开发者暴露概念和接口，只作为框架自身的逻辑抽象。TensorFlow 的开发者为了兼顾系统灵活性与高效性，在数据结构设计方面有不少别具匠心之处。理解这些设计，亦有助于读者增长开发技巧。

TensorFlow 中的部分数据结构定义在 Protocol Buffers 文件（*.proto）中，以便用于 gRPC 通信、检查点存储等有序列化需求的场景。它们会在代码构建时，由 Protocol Buffers 编译器生成对应的 C++ 源文件和头文件。如果在未编译的 C++ 源代码包中找不到某个类型的定义，不妨在 *.proto 文件中检索。

本节介绍 C++ 层面同张量、设备及数据流图相关的主要数据结构，这些数据结构在 TensorFlow 的多个组件中均有使用。专用于特定组件的其他数据结构将在相关章节中说明。

12.2.1　张量相关数据结构

张量作为一种基础数据表示模型，是 TensorFlow 的核心抽象，也是运行时内存的主要占用者和重点管理对象。出于支持异构设备内存、提高执行效率和节约存储开销等原因，TensorFlow 核心代码提供了多个与张量相关的数据结构，用于不同的代码场合。图 12-2 给出了与张量相关的主要数据结构的 UML 类图。

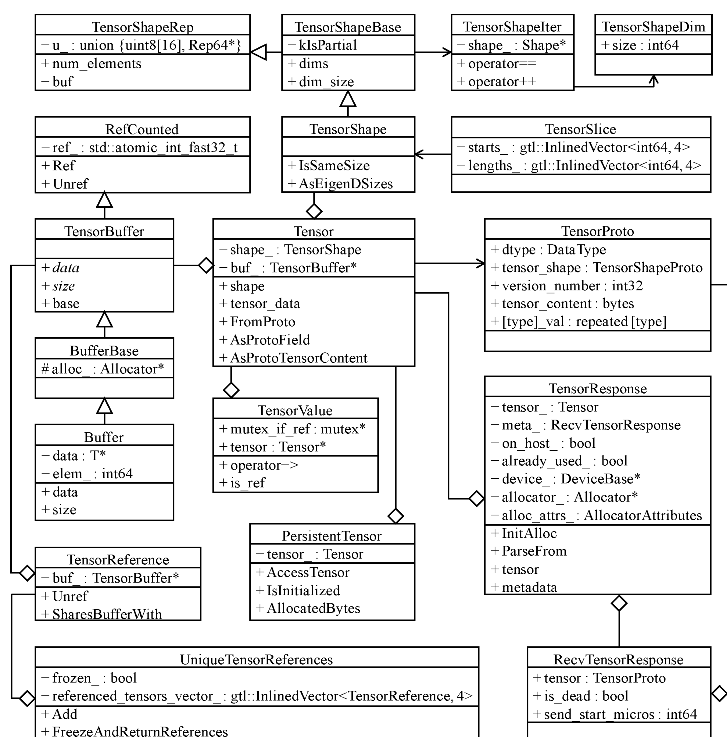


图 12-2　张量相关类的 UML 类图（只列出关键成员变量与方法）

Tensor 类是张量最为常用的抽象，它适用于 TensorFlow 运行时框架内部的多种场景，包括数据流图操作节点的输入/输出、设备无关的张量统一表示，以及仅存储元信息的临时变量等。Tensor 类的成员变量主要包括 TensorShape 对象和 TensorBuffer 指针，成员方法主要用于访问 TensorShape 和 TensorBuffer 的属性，以及将 Tensor 类型同其他相关类型相互转换。Tensor 类具有多种构造函数，支持将对象的元信息构造过程与缓冲区分配过程分离。

TensorShape 类用于记录张量的形状，即多维数组的数据类型、维数及各维大小。它具有名为 TensorShapeRep 和 TensorShapeBase 的两级父类。TensorShapeRep 类用于存储形状的信息，其中表示张量数据类型的 DataType 枚举定义于 types.proto 文件。为了既支持无限维度的张量，又避免动态内存管理开销，TensorShapeRep 类采取一种折中的存储方式：对于张量维数较小的情况，静态存储各维大小；对于维数较大的情况，使用封装了 std::vector 的 gtl::InlinedVector 对象动态存储各维大小。TensorShapeBase 类提供访问形状信息的接口，并以模板方式支持形状类的特化实现。它的辅助类型 TensorShapeIter 和 TensorShapeDim 提供对各维大小的遍历访问功能。同张量形状相关的还有 TensorSlice 类，它通过一组映射到每个维度的“起点-长度”整数对，描述一个形状在一组维度上的分片（slice）。该类主要用于检查点操作对张量局部存取的需求。

TensorBuffer 类用于管理存储张量数据的内存缓冲区。它继承了 RefCounted 类，具有引用计数能力。RefCounted 对象构造时，引用计数置为 1；调用 Ref 或 Unref 方法可将引用计数加 1 或减 1；引用计数为 0 时对象的析构函数将被调用。TensorBuffer 类提供 data 和 size 虚方法，用于取得缓冲区的常量指针及大小；另提供 base 方法，用于取得缓冲区的可写指针。TensorBuffer 的子类 BufferBase 及二级子类 Buffer 分别以非模板化和模板化方式实现了缓冲区管理，后者是 TensorFlow 代码中实际常用的类型。Buffer 类的 data_ 指针指向张量缓冲区，该缓冲区可以位于主机（CPU）内存或是设备（GPU）内存。Buffer 类的构造函数传入的 Allocator 及 AllocationAttributes 类型的参数用于指定缓冲区的内存分配器及内存属性。在使用重载过的赋值操作符复制 Tensor 对象时，内部的 TensorShape 对象和 TensorBuffer 指针得以复制，而 TensorBuffer 子类对象及其指向的缓冲区内容不会被复制，它们将在 Tensor 对象之间复用。引用计数机制使得缓冲区复用行为能够被精确记录，直到没有对象引用该缓冲区时才释放内存资源。

张量的另一种主要存储格式是 TensorProto 类，该类定义于 tensor.proto 文件。它主要用于基于 gRPC 通信库的进程间张量传输及相关场景。TensorProto 类的成员包括 DataType 对象、TensorShapeProto 对象、tensor_content 字节数组，以及十余种不同数据类型的重复型（repeated）Protocol Buffers 字段（相当于 C++ 数组或向量）。其中，表示形状的 TensorShapeProto 类同 TensorShape 类基本等价，但其存储形式易于序列化。字节数组和重复型字段是张量在 TensorProto 对象中两种互斥的表达形式，任何 TensorProto 对象只会在字节数组和十余种重复型字段中选择唯一一个字段存储张量数据。字节数组形式可以序列化存储任何类型的张量数据，具有通用和传输效率高的优势。重复型字段形式可以向量化存储 DataType 对应类型的张量

数据，具有易用性的优势。Tensor 类的 AsProtoTensorContent 和 AsProtoField 方法用于生成这两种表达形式的 TensorProto 对象。Tensor::FromProto 方法则相反，用于解析 TensorProto 数据，填充 Tensor 对象的缓冲区。TensorProto 对象始终保存在主存中，以便 gRPC 库直接访问。

与 gRPC 通信过程相关的还有 RecvTensorResponse 和 TensorResponse 类。RecvTensorResponse 类是定义在 worker.proto 文件中的、用于 gRPC RecvTensor 服务接口的响应消息类。它以 TensorProto 成员作为数据载体，同时记录了响应张量的一些属性。TensorResponse 类是 Tensor 类的封装形式，能够以高效的方式将保存在 gRPC 消息中的 TensorProto 对象反序列化为 Tensor 对象。TensorResponse 对象包含一个 Tensor 成员、一个用于保存张量属性的 RecvTensorResponse 成员，以及一组与设备和内存分配相关的指针。13.3.3 节将详述 TensorResponse 类的设计实现原理及其与 RecvTensorResponse 类的关系。

TensorReference 类用于持有 Tensor 对象的引用，从而保证在显式指定的周期之内，引用的 Tensor 对象始终不会被析构。它的唯一成员变量是 TensorBuffer 指针，其工作原理是在构造对象时增加 TensorBuffer 的引用计数，调用 Unref 方法时减少 TensorBuffer 的引用计数。TensorReference 类主要用在一些异步回调函数访问 Tensor 对象的场合。外层代码在回调函数对象定义时创建 TensorReference 对象，确保将来回调函数被调用时，直接或间接作为捕获参数的 Tensor 对象仍然可用。TensorFlow 另提供保存独一（unique）TensorReference 对象集合的 UniqueTensorReferences 类，该类型主要用于在操作上下文（OpKernelContext）对象中统一管理一组 Tensor 对象引用。

TensorValue 结构是另一种持有 Tensor 对象引用的类型，它主要作为数据流图上操作节点输入/输出张量的内部存储类型。该结构包含一个 Tensor 指针和一个互斥锁（mutex）指针，可以实现多个操作共享同一个 Tensor 对象，并且通过加锁方式安全地对同一 Tensor 对象实施读写。为操作节点提供安全访问张量缓冲区能力的数据结构还有 PersistentTensor 类。它封装 Tensor 对象而非指针，提供 AccessTensor、IsInitialized 等方法。操作节点通过该类访问张量时，AccessTensor 方法能够通知 OpKernelContext 对象，将张量的引用（即 TensorReference 对象）加入当前上下文的 UniqueTensorReferences 成员，从而确保张量在操作生命周期内的可用性。

12.2.2　设备相关数据结构

设备是对提供计算能力的硬件的抽象。在服务器端，TensorFlow 的开源版本目前支持 CPU 以及 CUDA、OpenCL 这两类编程模型的 GPU 作为计算设备。TensorFlow 核心代码提供一组层次化的数据结构，用于管理属性各异的设备。同时提供一些辅助数据结构，帮助设备管理机制与其他核心机制协同工作。图 12-3 给出了设备相关的主要数据结构的 UML 类图。

DeviceBase 类是所有设备类型的基类，主要功能是存储与设备相关的内部对象的指针。它的成员变量包括指向 Env、CpuWorkerThreads、GpuDeviceInfo、Eigen::ThreadPoolDevice 和 Eigen::SyclDevice 对象的指针，成员方法主要用于对上述变量进行访问。Env 类封装了与

操作系统相关的功能接口，包括文件系统访问、线程管理、动态库加载、计时等，用于通过设备访问与其关联的系统环境。`Env` 类的实现子类包括 `PosixEnv` 和 `WindowsEnv` 等。`DeviceBase` 类的其他成员变量用于保存不同类型设备所需的元信息指针。这些变量放置在基类而非子类中，与设备类的继承顺序设计相关，同时也方便部分函数直接使用基类指针操作具体子类。其中，`CpuWorkerThreads` 结构封装了 TensorFlow 内部定义的 `thread::ThreadPool` 类型指针，是 CPU 设备的线程池抽象，可将函数调度到 CPU 上执行。`GpuDeviceInfo` 结构封装了 `perftools::gputools::Stream`、`DeviceContext` 和 `EventMgr` 指针，用于管理 CUDA GPU 设备的执行流与事件，能够将计算任务加载到 GPU 上执行，以及实现跨设备的内存复制。`Eigen::ThreadPoolDevice` 结构是 Eigen 库对 CPU 设备的线程池抽象，在一部分计算类操作节点中使用。`Eigen::SyclDevice` 结构是 Eigen 库对 OpenCL GPU 设备的抽象，用于管理该类设备的计算任务与内存访问。

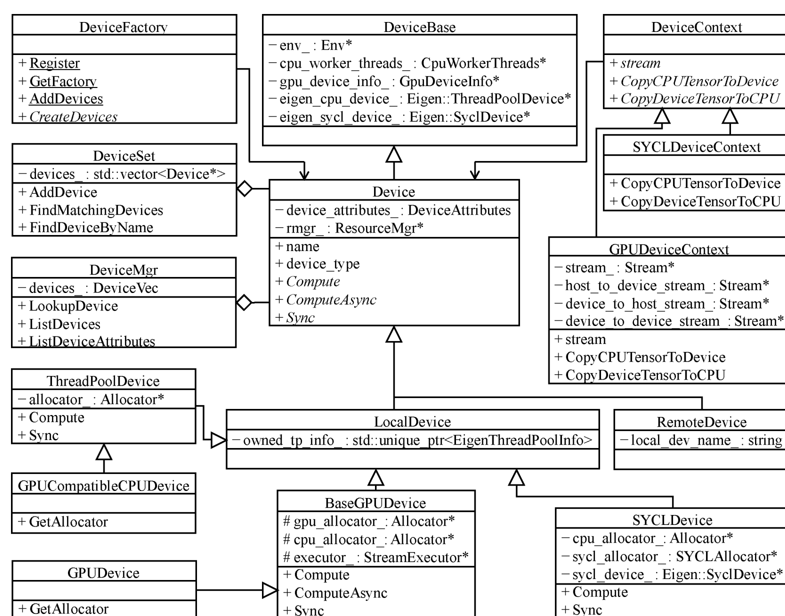


图 12-3　设备相关类的 UML 类图（只列出关键成员变量与方法）

`Device` 类继承自 `DeviceBase` 类，主要功能是存储设备本身的元信息，并为上层代码使用不同设备的计算能力提供统一的接口。它的主要成员变量包括设备属性类 `DeviceAttributes` 的对象，以及指向该设备关联的资源管理器 `ResourceMgr` 对象的指针。`DeviceAttributes` 类定义在 device_attributes.proto 文件中，用于记录设备的元信息，包括设备名称、类型、内存容量等。`ResourceMgr` 类以键值对方式记录与具体操作节点相关的、抽象为 `ResourceBase` 类型的逻辑资源，并提供资源的新增、删除、查找等接口。`Device` 类的主要成员方法包括访问设备元信息的 `name`、`device_type` 等方法，以及访问设备计算能力的 `Compute`、`ComputeAsync` 和 `Sync` 等虚方法。子类实现的 `Compute` 和 `ComputeAsync` 方法用于将操作节点提交到本设备上同步或异步执行，`Sync` 方法用于阻塞等待设备操作队列上的所有调用完成。

Device 类的子类 LocalDevice 和 RemoteDevice 分别是对本地设备和远程设备的抽象。本地和远程的概念以进程为界定单位，本进程注册的本服务器上的设备称为本地设备；本进程通过 gRPC 通信机制访问的其他进程注册的设备，无论是否和本进程位于同一台服务器，均称为远程设备。LocalDevice 类在 Device 类的基础上主要增加了 EigenThreadPoolInfo 指针成员，用于记录 Eigen 库线程池的元信息。LocalDevice 类的构造函数会对基类线程池相关成员进行初始化。RemoteDevice 类是远程设备在本地的“存根”（stub），它在 Device 类的基础上增加了不含位置前缀的本地设备名称字段。TensorFlow 提供 NewRemoteDevices 全局函数，用于创建 GrpcRemoteWorker 对象管理的所有远程设备对应的 RemoteDevice 对象集合。

ThreadPoolDevice 类（这里指 tensorflow::ThreadPoolDevice 类，而非 Eigen::ThreadPoolDevice 结构）是针对 CPU 设备的 LocalDevice 子类，实现了 Compute、Sync 等方法。ThreadPoolDevice::Compute 方法调用数据流图操作节点的 Compute 方法，同步执行相应操作。CPU 设备上的异步操作不通过 ThreadPoolDevice 类执行，而是使用 Eigen 库的线程池机制管理。BaseGPUDevice 类是针对 CUDA GPU 设备的 LocalDevice 子类，实现了 Compute、ComputeAsync、Sync 等方法。这些方法通过 StreamExecutor 库的流管理机制调度数据流图操作节点的执行。针对 CPU—GPU 内存复制等互操作需求，ThreadPoolDevice 和 BaseGPUDevice 类分别具有名为 GPUCompatibleCPUDevice 和 GPUDevice 的子类。它们实现了 GetAllocator 方法，能够按需获取适当的内存分配器。SYCLDevice 类是针对 OpenCL GPU 设备的 LocalDevice 子类，实现了 Compute 和 Sync 等方法。TensorFlow 开源版本现阶段对 OpenCL GPU 设备的支持相对有限，高级特性的支持程度不如 CUDA GPU 设备丰富。

在 TensorFlow 中，设备上下文抽象用于管理与设备当前执行状态相关的变量，并提供设备间互操作的方法。DeviceContext 类是设备上下文的基类，它定义了访问 GPU 设备流抽象的 stream 虚方法，以及用于 CPU—GPU 内存间复制的 CopyCPUTensorToDevice 和 CopyDeviceTensorToCPU 虚方法。CUDA GPU 的上下文子类是 GPUDeviceContext 类，它管理了 StreamExecutor 库提供的多种执行流，并基于这些流上的异步操作实现了设备内存间复制函数。OpenCL GPU 的上下文子类是 SYCLDeviceContext 类，它基于 Eigen 库封装的内存管理机制实现了设备内存间复制函数。

DeviceSet 类是设备集合的抽象。它以向量方式存储一组设备对象指针，提供在集合中增加、查找和枚举设备的方法，并支持设备按类型排序。DeviceSet 类主要用于数据流图操作节点调度场合。DeviceMgr 类是设备管理器的抽象，也可看作是一种设备集合类。它同样以向量方式存储一组设备对象指针，提供枚举、查找设备及列举所有设备属性的方法。在会话、会合点和进程间通信等场合，DeviceMgr 类用于统一管理相应机制关联的所有设备。

Device 类的对象实例创建采用工厂模式。Device 工厂的基类是 DeviceFactory 类。同 Device 类的继承结构类似，工厂类也有层次化的实现子类，包括 ThreadPoolDeviceFactory、BaseGPUDeviceFactory、GPUCompatibleCPUDeviceFactory、GPUDeviceFactory 和 SYCLDeviceFactory。DeviceFactory 类的静态方法 Register 和 GetFactory 用于注册和获取针对特定类型设备的工厂，静态方法 AddDevices 用于将本地所有满足特定约束的 Device 对象以指针向量形式输出。

AddDevices 方法内部调用不同设备工厂子类实现的 CreateDevices 方法，完成设备子类的对象实例创建。

另外，由于 GPU 设备是 TensorFlow 开源版本支持的主要加速器，TensorFlow 还提供了若干辅助数据结构，用于维护 GPU 设备的细节信息。这些数据结构定义在 perftools::gputools 命名空间，包括用于描述硬件参数、服务于 StreamExecutor 调度的 DeviceDescription 类，用于以位元组方式记录设备初始化选项的 DeviceOptions 结构，以及用于描述和管理设备内存的 DeviceMemoryBase、DeviceMemory 和 SharedDeviceMemory 类等。

12.2.3　数据流图相关的数据结构

在 TensorFlow 代码中，数据流图作为对计算过程的核心抽象，占有重要地位。数据流图相关的数据结构众多，涵盖了图的构建与执行的全生命周期，涉及计算关系的静态表示与计算流程的动态调度。图 12-4 首先给出了 TensorFlow 的 Protocol Buffers 协议定义中，与数据流图相关的主要数据结构的 UML 类图。这些数据结构大多定义在 tensorflow/core/framework 目录下的.proto 文件中。

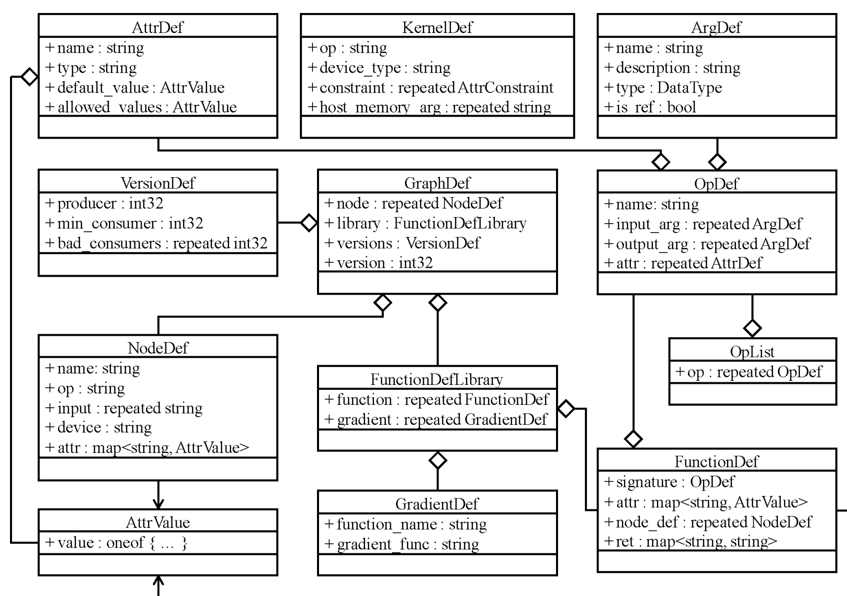


图 12-4　Protocol Buffers 协议定义中数据流图相关类的 UML 类图（只列出关键成员变量与方法）

在 Protocol Buffers 协议代码中，数据流图的数据结构为 GraphDef 类，它的关键成员是重复型 NodeDef 类型的 node 字段，即数据流图所包含的节点列表。该类有两个表示序列化数据格式版本的成员，分别为 VersionDef 类型的 versions 和 int32 类型的 version，二者用于在多个 TensorFlow 版本共存的分布式环境中，确保数据格式的兼容性。前者是 TensorFlow 主推的“语义化版本”（Semantic Versioning，具体定义可参见 http://semver.org/ ）表示方式，后者仅用于与

TensorFlow 早期版本兼容。有关序列化数据格式的版本演进历史，可参考 tensorflow/core/public/version.h 文件对 TF_GRAPH_DEF_VERSION 宏的注释。此外，该类还包含 FunctionDefLibrary 类型的 library 字段，用于记录用户自定义的函数。

表示操作的数据结构为 OpDef 类，该类的对象表达的是一类特定操作的共有特征，而非一幅数据流图上的某个具体操作节点。该类的主要成员包括 string 类型的 name、重复型 ArgDef 类型的 input_arg 和 output_arg，以及重复型 AttrDef 类型的 attr 等。name 成员定义操作的名称，具体可参考 tensorflow/core/ops 目录下操作相关代码调用 REGISTER_OP 宏注册操作时所定义的名称。ArgDef 类用于描述操作的输入或输出参数，其成员定义了参数的名称和数据类型等信息。AttrDef 类用于描述操作所具有的属性，其成员定义了属性的名称、数据类型和默认值等信息。OpDef 类的集合类型是 OpList 类。

表示操作对应的算法核函数的数据结构为 KernelDef 类。一个 KernelDef 实例由一种操作、一种设备及一组约束条件确定。string 类型的 op 和 device_type 字段指示核函数对应的操作名称及设备类型，重复型 AttrConstraint 类型的 constraint 成员给定操作属性的取值范围约束，重复型 string 类型的 host_memory_arg 成员指定需要保存在主机内存上的操作参数名称。

数据流图节点的数据结构为 NodeDef 类。其中，string 类型的 name、op 和 device 成员分别记录节点的名称、操作名和设备名。重复型 string 类型的 input 成员记录本节点连接的输入节点。map 类型的 attr 成员以键值对方式记录与特定操作相关的属性集合。节点名称依据操作的不同会有不同的格式，同一幅数据流图中的节点名称不能重复。在作为 input 成员元素时，节点名称可以选择性地添加“:[输出张量序号]”后缀，用于说明输入节点的哪个输出张量连接到本节点。操作名的取值是操作对应的 OpDef 对象的 name 值，同种操作的名称相同。设备名的一般格式为“@[主机名]/job:[作业名称]/replica:[副本 ID]/task:[任务 ID]/[cpu|gpu]:[设备 ID]”。属性集合 attr 的键类型为 string,其取值为属性在 OpDef 中对应的 AttrDef 对象的 name 值。attr 的值类型为 AttrValue，该类通过 Protocol Buffers 的 oneof 型字段（类似于 C/C++ 的 union）支持多种不同类型的属性值。

截至 TensorFlow 1.2 版本，数据流图上的自定义函数仍是一个实验性特性，用于封装并复用一组以固定模式组合的操作。如果图中某个 NodeDef 对象的 op 值不是 TensorFlow 固有的 OpDef 类所定义的操作名称，而是 GraphDef 对象的 library 字段所管理的自定义函数名称，那么当该节点的输入就绪时，自定义函数会被调用。自定义函数由 FunctionDefLibrary 类管理。FunctionDefLibrary 类只有两个成员，即重复型 FunctionDef 类型的 function 和重复型 GradientDef 类型的 gradient。FunctionDef 类用于描述自定义函数的元信息，GradientDef 类用于记录自定义函数对应的梯度计算函数。

接下来关注 TensorFlow 核心的 C++ 代码中的数据结构定义。相比于主要用于元信息存储和序列化表示的 Protocol Buffers 数据结构，C++ 类型定义更注重对数据的操作，以便于数据流图执行逻辑的高效实现。图 12-5 给出了 C++ 代码中与数据流图相关的主要数据结构的 UML 类图。

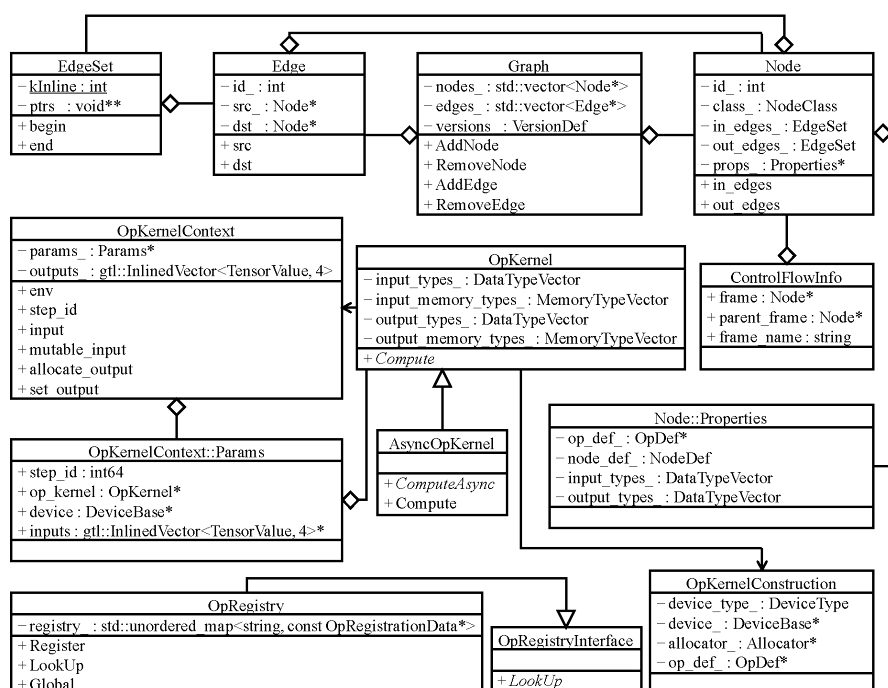


图 12-5 C++ 代码中数据流图相关类的 UML 类图（只列出关键成员变量与方法）

在 C++ 代码中，数据流图的数据结构为 Graph 类，图上的节点和边分别为 Node 类和 Edge 类。Graph 类的关键成员变量包括节点指针向量 nodes_、边指针向量 edges_、VersionDef 类型的数据格式版本字段 versions_ 等，成员方法主要用于增加、删除及查找节点和边。Node 类的主要成员变量包含标识符 id_、类型枚举 class_、输入和输出边集 in_edges_ 和 out_edges_ 等，并通过其嵌套的 Properties 类封装了 NodeDef 对象及 OpDef 指针。Edge 类的主要成员变量包含标识符 id_、源与目标节点指针 src_ 和 dst_ 等。这两个类的成员方法大多用于访问各自对象的成员变量。Node 的辅助类 NodeBuilder 用于以方法链（method chaining）语法便捷地构造 Node 对象。Edge 的辅助类 EdgeSet 通过内置数组与 std::set 对象结合的方式高效存储边的集合。为了实现数据流图执行的时序控制，TensorFlow 引入了控制流的概念，使用 ControlFlowInfo 结构记录控制流的信息。

针对数据流图操作的高效管理需求，TensorFlow 提供了与 OpDef 相关的多种工具类。接口类 OpRegistryInterface 及其实现类 OpRegistry 和 OpListOpRegistry 用于注册和查找 OpDef 对象。其中，OpRegistry 类是运行时库全局的、单例（singleton）的 OpDef 管理器，它的 Global 方法返回其唯一实例。OpListOpRegistry 类则是一种基于已有 OpList 对象构造的 OpDef 管理器。OpDefBuilder 类用于以方法链语法构造 OpDef 对象。OpDefBuilder 的包装类 OpDefBuilderWrapper 以模板特化方式提供选择性注册 OpDef 对象的功能。开发者可以编写 ops_to_register.h 头文件，通过 SHOULD_REGISTER_OP 宏指定某个操作是否需要被注册，以便减小 TensorFlow 核心的二进制

文件体积。辅助结构 OpDefBuilderReceiver 用于将 REGISTER_OP 宏注册 OpDef 对象的请求转发给 OpRegistry 对象处理。

OpKernel 类是操作对应的核函数基类。它包含用于记录输入、输出边数据类型和内存类型的向量成员。其关键方法是用于同步计算的 Compute 虚方法。OpKernel 的子类 AsyncOpKernel 是异步核函数的基类，提供用于异步计算的 ComputeAsync 虚方法。具体的核函数子类需要实现 Compute 或 ComputeAsync 方法。辅助类 OpKernelConstruction 作为 OpKernel 构造函数的参数，用于保存构造 OpKernel 对象时所需的所有元信息。工具类 OpKernelRegistrar 类用于在系统全局注册核函数对象，注册信息以单例的 KernelRegistration 结构映射表形式保存。为便于编写注册代码，KernelRegistration 类及 REGISTER_KERNEL_BUILDER 宏提供了一套基于方法链语法的封装。另有辅助类 OpSegment，用于维护一个会话中注册的 OpKernel 对象集合。

OpKernelContext 类用于记录操作核函数对象的上下文。OpKernel、AsyncOpKernel 的 Compute、ComputeAsync 方法均具有上下文指针参数，用来将核函数对象与一幅数据流图上的某个具体操作节点关联起来，进而能够对节点实例执行计算或通信操作。OpKernelContext 类的成员众多，其中作为构造函数参数使用的成员变量封装在名为 Params 的内嵌结构中。这些成员变量包括输入/输出 TensorValue 向量、执行步 ID、OpKernel 和 DeviceBase 指针等。OpKernelContext 类的成员方法提供多种适用于不同场景的成员变量访问接口，包括只读访问与可写访问接口等，旨在实现内存复用。

Protocol Buffers 数据流图可通过 GraphConstructor 类生成对应的 C++ 数据结构。GraphConstructor 类的核心方法是 Construct，它使用 GraphDef 对象存储的信息构造 Graph 对象。辅助数据结构 NodeInfo 和 EdgeInfo 等用于在这一过程中实现信息映射。全局函数 ConvertGraphDefToGraph 是上层代码访问数据结构转换逻辑的主要接口。

12.3 公共基础机制

TensorFlow 提供一系列公共基础机制，它们服务于多种不同的功能模块。理解这些机制的原理与设计，有助于进一步理解各个功能模块。本节选取并介绍内存分配、线程管理、多语言接口、XLA 编译技术和单元测试框架等具有代表性的公共基础机制。

12.3.1 内存分配

对于机器学习、深度学习算法执行而言，内存的容量及访问速率是影响计算性能的重要因素。平台层软件能否有效地使用内存资源，将其高效地分配给计算任务，很大程度上决定了软件的整体性能。TensorFlow 并没有简单地利用标准库的动态内存分配机制，而是设计了一组内存分配器组件。这些内存分配器一方面针对不同设备和工作场景，实现了相对高效的内存分配方法；另一方面屏蔽了异构设备内存分配接口的差异，提升了核心组件的易用性。图 12-6 给出了 TensorFlow 提供的主要内存分配器类型及其继承关系。

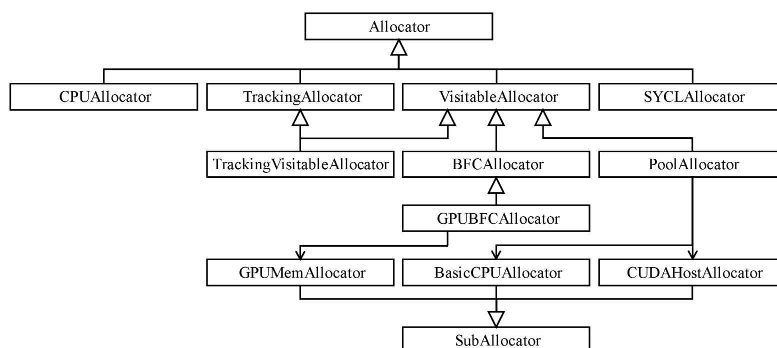


图 12-6 主要内存分配器类型及其继承关系

`Allocator` 类是内存分配器的基类，用于以一致的接口分配或回收不同设备的内存空间。它提供一组虚方法，包括用于分配内存的 `AllocateRaw` 和回收内存的 `DeallocateRaw` 等，并出于易用性的目的，封装了多种重载版本的 `Allocate`、`Deallocate` 方法及其他辅助方法。`Allocator` 类具有以下两种属性抽象。

- `AllocationAttributes` **结构**：用于描述单次内存分配的属性，目前提供的字段包括支持失败重试和分配行为记录的开关变量，它主要作为内存分配方法的参数使用。
- `AllocatorAttributes` **结构**：用于描述一种内存分配器固有的属性，它以位元组方式标识了某种分配器管理的内存是否位于主存、是否具有 GPU 兼容性、是否能够跟踪分配大小等特性。该结构主要作为分配器子类的工厂方法参数，或其他组件记录自身管理内存属性的字段。

`Allocator` 类还具有两种包装器子类：`TrackingAllocator` 子类为内存分配器提供已分配空间大小的记录功能，`VisitableAllocator` 子类为内存分配器提供针对特定访问者的区块注册功能。另有 `TrackingVisitableAllocator` 类，它继承了这两种包装器的能力。

为了便于组合不同内存分配器的功能，同时避免多重继承的开销，TensorFlow 设计了 `SubAllocator` 类型。该类仅提供最基本的 `Alloc` 和 `Free` 虚方法。其子类实现这些方法之后，可以被其他复杂的 `Allocator` 子类作为成员对象集成。`SubAllocator` 类现有的主要子类包括用于分配 CPU 内存的 `BasicCPUAllocator` 类、用于分配 CUDA GPU 内存的 `GPUMemAllocator` 类，以及用于在主存上分配锁页（pinned）内存的 `CUDAHostAllocator` 类。

针对具体的计算设备，TensorFlow 提供了不同的 `Allocator` 子类。`CPUAllocator` 类实现了面向主机（即 CPU 计算设备）的内存分配器，它的内存分配与回收方法在内部调用 `tensorflow::port` 命名空间下的 `AlignedMalloc` 和 `AlignedFree` 方法。`tensorflow::port` 命名空间针对不同操作系统具有不同的实现代码。以 Linux 系统为例，内存分配器的具体实现允许在 TensorFlow 源代码编译时加以选择——可以选择来源于 FreeBSD 社区、被多种网络服务软件使用的 jemalloc（详见 http://jemalloc.net），也可以选择操作系统 C/C++ 标准库提供的默认内存分配器（一般为

ptmalloc，详见 http://malloc.de/en）。全局函数 cpu_allocator 用于返回经过 TrackingAllocator 包装的单例 CPUAllocator 对象。

GPUBFCAllocator 类实现了面向 CUDA GPU 的内存分配器，该分配器依次继承了 BFCAllocator 和 VisitableAllocator 父类。BFCAllocator 类实现了 BFC（best-fit with coalescing）算法（详见 http://g.oswego.edu/dl/html/malloc.html）——这是一种经典的内存分配算法，能够通过内存的分块与合并实现高效的空间分配和碎片回收。GPUBFCAllocator 类在继承父类算法的同时，将内存分配的具体实现绑定到 GPUMemAllocator 对象。后者通过访问 StreamExecutor 库的相关接口实现 GPU 内存分配与回收。SYCLAllocator 类实现了面向 OpenCL GPU 的内存分配器。它的逻辑比较简单，直接调用了 Eigen 库的对应方法。TensorFlow 还提供面向 CPU——GPU 内存 DMA 通信场景、针对 CPU 内存分配优化的 PoolAllocator 类，该类的实现借助了 BasicCPUAllocator 和 CUDAHostAllocator 对象。

12.3.2　线程管理

TensorFlow 运行时核心引入了多线程设计，充分发挥多核 CPU 的计算潜力，利用系统的并行处理能力提升软件性能。TensorFlow 的线程管理机制与很多数据计算平台类似，以动态、异步、事件驱动的任务并行为主要特征，具有高度的灵活性。

为了简化多线程程序开发，同时节省线程创建和销毁的开销，TensorFlow 采用经典的线程池模式管理多线程。线程池类 ThreadPool 定义于 tensorflow/core/lib/core/threadpool.h 文件。该类最基本的构造函数签名为 ThreadPool(Env* env, const string& name, int num_threads)，其中 name 和 num_threads 参数指定线程池的名称与大小。Env 类包含与特定操作系统相关的系统调用封装。将 env 指针传入 ThreadPool 对象的目的是获取当前操作系统的线程创建方法（封装于 Env::StartThread）。Schedule 方法用于将函数对象提交到线程池以待调度执行，其签名为 Schedule(std::function<void()> fn)。该方法是非阻塞的，函数对象的执行时机由线程池的内部实现决定。ThreadPool 类还提供并行执行一组相同函数的 ParallelFor 方法，其语义类似于 OpenMP 的 parallel for 操作，一般用于数据并行的计算模式。

ThreadPool 类通过内嵌的 Impl 结构封装了 Eigen 库的线程池实现。Impl 结构继承自 Eigen 库的 ThreadPoolTempl 类模板，并通过 EigenEnvironment 结构对模板进行特化。Eigen 库的 ThreadPoolTempl 类模板提供先入先出（SimpleThreadPoolTempl）和非阻塞队列（NonBlocking-ThreadPoolTempl）两种实现。池内有线程空闲时，空闲线程将从提交队列中加载一个函数对象执行。EigenEnvironment 结构的作用是将 TensorFlow 定义的线程抽象及操作按照 Eigen 库要求的接口进行封装，以便 ThreadPoolTempl 类模板调用。在针对类 Unix 和 Windows 操作系统的 Env 子类中，内部封装的线程实现均是 C++ 的 std::thread。

在 TensorFlow 应用实例运行时，每个进程具有一个名为 Compute 的全局线程池对象，该对象可通过定义于 tensorflow/core/common_runtime/process_util.h 文件的全局函数 ComputePool 访

问。Compute 线程池的初始化是在首次访问时由全局函数 InitComputePool 执行的，线程池的大小为进程可调度的 CPU 总核数。进程内的 worker 服务对应的 WorkerEnv 对象持有 Compute 线程池的指针。worker 服务收到 RPC 请求时，将相应的方法实现函数调度到 Compute 线程池内执行。此外，DirectSession 类中包含一个线程池向量，这些线程池用于执行数据流图上的操作。

除了支持线程池调度模式外，TensorFlow 运行时核心也允许通过直接新建线程的方式执行函数对象（闭包）。这种方式的接口是 tensorflow/core/common_runtime/process_util.h 文件中定义的 SchedClosure 全局函数。该函数内部调用当前操作系统 Env 子类的 SchedClosure 方法。在 TensorFlow 当前的实现中，Env 子类的 SchedClosure 方法新建 std::thread 对象执行函数，随后将新线程与主线程分离（detach）。这种设计虽然有一定性能和内存开销，但可以避免大量函数对象在线程池中阻塞等待。这种方式的典型用例是 Master 类的一些成员方法：这些方法将会话相关操作的收尾工作封装到匿名函数中，以新建线程方式执行，这样能够加速主线程中方法的返回。

12.3.3　多语言接口

考虑到系统运行时的高效性，TensorFlow 以 C++ 作为核心实现语言。为了兼顾开发效率并降低学习成本，TensorFlow 在核心库之上为应用开发者提供 C++、Python、Java、Go 等多种语言的 API，社区亦提供 Haskell、Rust 等其他语言的 API。为了简化多语言 API 的实现，TensorFlow 具有一套以 C 语言为中介的接口机制。在 tensorflow/c 目录下，c_api.h 和 c_api.cc 文件通过一组名称以 TF_ 开头的全局函数封装了 C++ 核心的主要功能，并以兼容 C 语言的命名方式将函数导出（extern "C"）。Python、Java、Go 等 API 借助这一接口访问 C++ 核心功能。C++ API 则直接封装 C++ 核心库的抽象与方法，无须经由 C API 转发。

Python API 与 C 语言接口的链接通过 SWIG 工具实现。SWIG 工具可依据扩展名为.i 的接口描述文件自动生成 Python 访问 C 的封装层代码。TensorFlow Python API 的顶层接口描述文件是 tensorflow/python/tensorflow.i，它引入了若干子模块的接口描述文件。这些接口描述文件或是直接引用 c_api.h 及其他核心头文件导出的函数，或是对核心功能进行进一步封装。使用源代码构建 TensorFlow 时，自定义的 Bazel 编译规则——tf_py_wrap_cc 调用 SWIG 工具和 C++ 编译器，将 C++ 核心代码编译到名为_pywrap_tensorflow_internal.so（以 Linux 系统为例）的动态链接库，同时生成名为 pywrap_tensorflow.py 和 pywrap_tensorflow_internal.py 的 Python 接口文件，以便上层 Python API 调用。

Java API 与 C 语言接口的链接通过 JNI（Java Native Interface，Java 本地接口）机制实现。TensorFlow 源代码包的 tensorflow/java/src/main/native 目录包含 JNI 工具链生成的头文件及相应的 C++ 实现文件，这些 C++ 文件中的函数实现封装了 c_api.h 提供的函数。tensorflow/java/src/main/java 目录则包含 TensorFlow 主要抽象的 Java 定义及其与 native 实现之间的胶合层代码。依据 Bazel 编译规则，native 目录中的代码连同必要的 C++ 核心代码编译生成名为 libtensorflow_jni.so（以 Linux 系统为例）的动态链接库，java 目录中的代码编译生成名为 libtensorflow.jar 的归档文件。在

Java API 代码中，声明为 native 的函数直接使用动态链接库中的函数实现，进而访问 C++ 核心功能。

Go API 与 C 语言接口的链接通过 cgo 机制实现。在 tensorflow/BUILD 文件中，TensorFlow 提供将 C API 连同 C++ 核心代码编译为标准动态链接库的能力。以 Linux 系统为例，生成的库文件名为 libtensorflow.so。Go API 代码中通过 `import "C"` 方式引入 C API 中的函数定义，并在应用程序代码编译时链接到上述动态链接库，实现 Go 应用程序对 TensorFlow 核心的访问。

12.3.4　XLA编译技术

从 TensorFlow 1.0 版本开始引入的 XLA（Accelerated Linear Algebra，加速线性代数）是一套通过编译技术优化机器学习平台性能的组件。其设计目标包括：加速数据流图执行，提升内存使用效率，降低自定义操作依赖，减小移动应用内存占用，以及增强平台可移植性。XLA 主要由数据流图转换器、XLA 编译器、JIT（just-in-time）编译机制、AOT（ahead-of-time）编译机制等模块构成。XLA 是对 TensorFlow 原有核函数及运行时组件的增强，用户可以选择性地开启。图 12-7 给出了 XLA 的主要模块在数据流图执行流程中的逻辑位置。

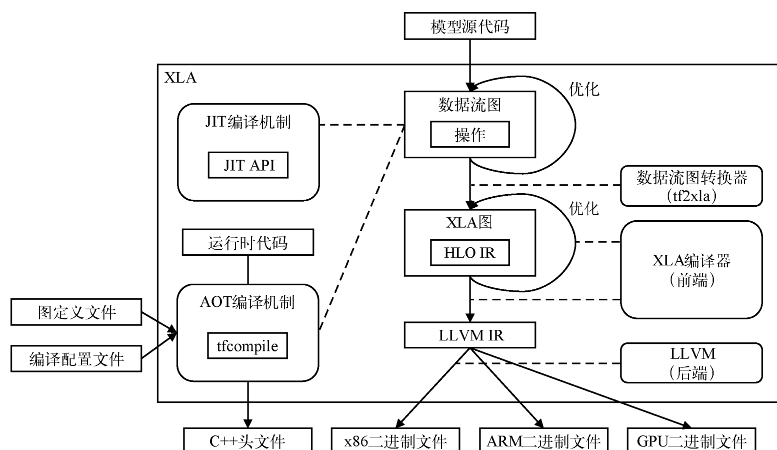


图 12-7　XLA 的主要模块在数据流图执行流程中的逻辑位置

XLA 技术的核心抽象是 HLO（High Level Optimization）中间表示层。HLO 是一种面向线性代数语义的编译器 IR（Intermediate Representation），由 HLO 表达的语法树亦称为 XLA 图。TensorFlow 官网或源代码包中的 operation_semantics.md 文档给出了 HLO 的操作语义详细说明。HLO 的引入将核函数开发时的前后端代码解耦，有助于增强 TensorFlow 的可移植性。XLA 技术的总体流程是将 TensorFlow 数据流图转换为 XLA 图，再由基于 LLVM 的编译器生成设备特定的二进制文件。

数据流图转换器用于将 TensorFlow 数据流图转换为 XLA 图，其源代码位于 tensorflow/compiler/tf2xla 目录。该模块定义了名为 `XlaOpKernel` 的 XLA 层核函数类型，并且实现了一系列线性代数相关的核函数子类。在数据流图执行前，转换器基于数据流图的结构创建 XLA 图，将原图中那些可通过 XLA 优化的操作节点替换为 XLA 层核函数。由于数据流图操作同 HLO IR 存

在语义差异，节点的替换可能是一对一的，也可能是一对多的。

XLA 编译器用于将 XLA 图编译为二进制文件，其源代码位于 tensorflow/compiler/xla 目录。该模块基于 LLVM 技术实现，目前支持的后端设备包括 x86 CPU、ARM CPU 和 NVIDIA GPU。在编译器中，前端解析并优化 XLA 图，将其底层化（lower）为 LLVM IR，后端将 LLVM IR 编译为设备特定的二进制文件，并执行二进制代码优化。XLA 编译器采用客户端-服务器模式设计，以便管理编译过程的生命周期。HLO 的实现亦位于该模块。

JIT 编译机制用于在 TensorFlow 应用运行时创建执行数据流图操作的二进制代码，它是一套面向异构设备的通用性能优化机制。该模块的源代码位于 tensorflow/compiler/jit 目录。模型开发者使用 JIT 机制时，需要在代码中显式引入 JIT API，将会话或操作配置为 XLA 编译模式。JIT 编译机制会对数据流图的可优化部分实施优化，将其中大量细粒度的操作融合（fuse）为少量粗粒度的专用核函数。这些核函数经由 XLA 编译器生成高效的二进制代码，能够减少图执行过程中内存分配和上下文切换开销。

AOT 编译机制用于在 TensorFlow 应用运行前创建集成了模型和运行时的二进制代码，主要适用于手机等移动端的推理态（inference）性能优化。该模块源代码位于 tensorflow/compiler/aot 目录。模型开发者使用 AOT 机制时，需要提供 Protocol Buffers 格式编写的数据流图定义文件及编译配置文件。然后使用 Bazel 构建工具，调用 AOT 封装工具——tfcompile 编译数据流图。AOT 编译机制生成的二进制代码包含了模型和必要的运行时逻辑，不再需要完整的 TensorFlow 运行时库支持，因此能够减小可执行程序的体积。一并生成的 C++ 头文件用于在应用代码中访问模型。

12.3.5　单元测试框架

TensorFlow 作为一个代码量达数十万行的项目，在软件工程管理方面较为规范和严谨。单元测试作为保证软件正确性的重要工具，自然是 TensorFlow 源代码包不可或缺的组成部分。TensorFlow 的单元测试框架使用了 Google 公司开源的生态系统组件，包括用于测试用例开发的 Google Test 工具，以及用于测试目标构建的 `bazel test` 命令。

TensorFlow 单元测试的入口代码位于 tensorflow/core/platform/test_main.cc 文件，相关辅助函数位于同一目录下的 test.h 和 test.cc 文件。各个组件的测试用例代码一般位于组件本身所在的目录中，文件名为“[组件名]_test.cc”。在这些文件中，测试用例类通过 `TEST` 宏定义，测试断言语句通过 `ASSERT` 或 `EXPECT` 系列宏封装。开发者运行特定测试时，只需要执行 `bazel test [组件的测试目标名]` 命令，例如：`bazel test //tensorflow/core:common_runtime_direct_session_test`。如果希望运行 TensorFlow 提供的所有单元测试，则可以执行 `bazel test //tensorflow/...`。

单元测试代码有助于读者学习 TensorFlow 的编程开发方法。其中，cc、python 等应用层 API 目录下的测试用例适用于算法模型和应用程序开发者学习上层抽象与接口的用法，而 core 目录下的测试用例则是系统平台层开源贡献者或二次开发者学习 TensorFlow 核心机制用法与原理的良好教程。开发者在编写新组件时，亦可模仿原有的测试用例编写自己的测试用例，然后运行所有

测试用例，以便验证新组件的正确性及其对原有组件的影响。例如，在设计一种新的内存分配算法时，开发者可以参考 allocator_test.cc 文件，了解内存分配器类需要实现的最小方法集合；通过将原有测试用例中的 `CPUAllocator` 对象替换为新的内存分配器，开发者可以快速验证新算法的正确性。

12.4　外部环境接口

TensorFlow 作为一套以性能为重要目标、面向实用场景的深度学习系统，充分考虑了与外部软硬件环境的对接。一方面，TensorFlow 需要支持以 GPU 为代表的加速器，以便利用日益增长的硬件能力，实现计算高效性；另一方面，它需要与多类操作系统及开源生态中的组件协同工作，支持云计算与移动终端等多样化的运行时环境，实现平台可移植性。

理解 TensorFlow 对外部软硬件组件的支持方式，有助于二次开发者及系统集成开发者充分利用 TensorFlow 软件架构的灵活性，拓展其生态系统，使之在更为广泛的场景下发挥效用。

12.4.1　加速器硬件接口

在高性能计算与大数据处理领域，GPU 是应用最为广泛的计算加速器硬件。GPU 针对向量和矩阵的并行计算能力恰恰是神经网络等算法的核心需求，因此 GPU 成为包括 TensorFlow 在内的绝大多数机器学习、深度学习系统的首选硬件加速器。TensorFlow 开源版本最初支持使用 CUDA 编程模型的 NVIDIA GPU，自 0.12 版开始亦实验性地支持使用 OpenCL 编程模型的其他 GPU。除了 GPU，Google 公司内部也开发了名为 TPU（Tensor Processing Unit）的定制化 ASIC 设备，作为服务器端的硬件加速器。但 TPU 不对外销售，TensorFlow 的开源版本也未提供相应开发接口。

在 CUDA GPU 支持方面，TensorFlow 并未使用常规的 CUDA 运行时 API 来访问 GPU，而是使用 StreamExecutor 库实现对 GPU 的低层次、细粒度控制。StreamExecutor 是 Google 公司开发的一套异构并行计算库，为 CUDA 和 OpenCL 编程模型的 GPU 提供主机端开发的统一接口及编译时和运行时支持。StreamExecutor 以“流”（stream）为基本编程抽象，以调度核函数（kernel）在流上的执行为主要工作方式，旨在提供一种安全、高效、易用的 GPU 并行编程模式。

TensorFlow 开源版本集成的 StreamExecutor 模块（位于 tensorflow/stream_executor 目录）是 StreamExecutor 项目的子集，重点支持 CUDA GPU。图 12-8 给出了该版本 StreamExecutor 的主要组件逻辑结构，以及它与上下层组件的关系。StreamExecutor 底层是设备适配层，包含对 CUDA 设备和主机（CPU）设备的接入逻辑。它将 CUDA 和主机原有的编程抽象映射到流抽象，并实现设备特有的内存管理函数。CUDA 适配层采用动态函数加载方式，直接调用 CUDA 驱动 API 访问 GPU。由于 CUDA 原生提供用于并发任务调度的流抽象，StreamExecutor 直接对其进行了封装。CUDA 适配层还接入了 cuBLAS、cuDNN 等函数库，以利于上层通用算子的高效实现。主机适配层封装了本地操作系统的内存分配等系统调用，并提供 HostStream 抽象，用于主机与 GPU 协同工作时安全高效地管理主机内存。适配层之上的 StreamExecutor 运行时引擎主要负责在流上调度执行核函数，它通过适配层封装的底层机制保证并发执行的安全性。StreamExecutor 对上层

开发者提供的 API 主要包含在 `Stream`、`Kernel`、`Timer`、`Event` 等类中。上层代码由此可以绕过 CUDA 运行时 API，使用这种对异步、并发、多设备编程更加友好的 API 编写 GPU 程序。在这些接口中，`Stream` 类是上层代码访问 GPU 的主要入口，它提供一系列名称以 `Then` 开头的计算与调度函数，用于异步加载执行核函数，并实现多种同步等待语义。为了简化数据计算系统，特别是深度学习系统的开发，StreamExecutor 还提供了一组通用算子库，包括用于基本线性代数的 BLAS、用于深度神经网络的 DNN、用于快速傅里叶变换的 FFT，以及用于随机数生成的 RNG 等。这些通用算子的调用接口亦被集成到了 `Stream` 类中。

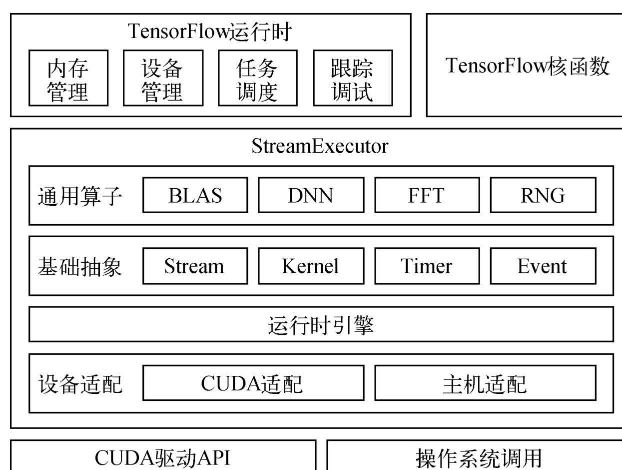


图 12-8　StreamExecutor 的主要组件逻辑结构

TensorFlow 对 CUDA GPU 的支持主要体现在运行时库和算法核函数这两方面。在 TensorFlow 核心的通用运行时库中，与 CUDA GPU 相关的代码位于 tensorflow/core/common_runtime/gpu 目录。运行时库封装了 StreamExecutor 的接口，提供了一组与 TensorFlow 系统层语义相关联的数据结构与实用工具类。这些代码主要涉及内存管理、设备管理、任务调度和跟踪调试等。内存分配器 `GPUBFCAllocator`、`CUDAHostAllocator` 等已在 12.3.1 节中介绍过。为了实现跨设备的内存复制，工具类 `GPUUtil` 提供了一组用于 `Tensor` 对象的内存复制方法。设备抽象 `BaseGPUDevice`、`GPUDevice` 等已在 12.2.2 节中介绍。设备对象通过 `StreamGroup` 结构封装的流指针访问 StreamExecutor 的接口。对于任务调度，运行时库提供了 `EventMgr` 类，它具有在流上调度执行异步函数的 `ThenExecute` 方法，主要用于在 `Tensor` 复制逻辑中维护对象的生命周期。对于跟踪调试，运行时库提供了具有调用统计和计时功能的 `GPUTracer` 类，以及用于内存分配器调试的 `GPUDebugAllocator` 类。在 TensorFlow 操作对应的核函数实现代码中，`OpKernelContext` 对象的 `DeviceContext` 成员为核函数提供 `stream` 方法，用于访问特定操作在运行时绑定的 GPU 设备对应的 `Stream` 对象。核函数通过调用 `Stream` 对象提供的计算方法，将计算任务加载到 GPU 上执行。

在 OpenCL GPU 支持方面，TensorFlow 主要利用 SYCL 接口（详见 https://www.khronos.org/sycl），而非以传统方式编写 OpenCL 核函数文件。SYCL 是一套基于 OpenCL 的“单源”（single-

source）异构编程机制，旨在提升 OpenCL 接口易用性和应用开发效率。所谓“单源”，是指主机和设备源代码处于同一个 C++ 文件中，由 SYCL 专用编译器借助 C++ 模板函数等机制为异构硬件同时生成二进制文件。SYCL 在继承 OpenCL 原有抽象的同时，还提供一系列高级的并发控制结构，能够简化异构并行应用的开发。图 12-9 给出了 SYCL 单源开发模式与 OpenCL 传统开发模式的对比，其中 SPIR（Standard Portable Intermediate Representation）是 OpenCL 的一种中间表示层语言。

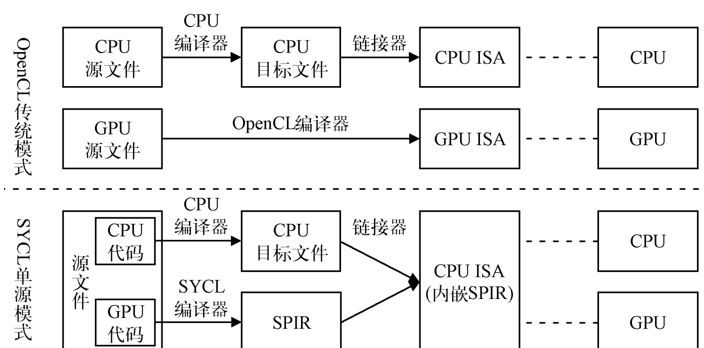


图 12-9　SYCL 单源开发模式与 OpenCL 传统开发模式

TensorFlow 的 SYCL 支持主要由 SYCL 技术的维护者——Codeplay 公司贡献。与集成在 TensorFlow 源代码包中的 StreamExecutor 模块不同，SYCL 开发包是一个外部依赖项。TensorFlow 对 OpenCL GPU 的支持同样体现在运行时库和算法核函数两方面。在通用的运行时库中，与 OpenCL GPU 相关的代码位于 tensorflow/core/common_runtime/sycl 目录。这里定义了内存分配器 `SYCLAllocator`、设备抽象 `SYCLDevice`、设备上下文 `SYCLDeviceContext` 等类型，实现了 TensorFlow 内存和设备管理的必要接口。截至 TensorFlow 1.2 版本，SYCL 核函数的实现尚不如 CUDA 核函数那样完善，目前主要提供逐系数（coefficient-wise）操作的核函数。运行时库与算法核函数的内部实现均基于 `Eigen::SyclDevice` 类型及其相关的计算组件——这些数据结构与算法是 Eigen 库集成的第三方包 Tensor 的一部分，它们的主要贡献者亦是 Codeplay 公司。

同一种数据流图操作之所以能够在不同类型的设备上执行，是因为其核函数针对不同设备进行了实现。为了简化异构设备上的核函数开发，TensorFlow 在操作相关的代码中利用了 C++ 模板编程技巧，并且部分地借助了 Eigen 库既有的异构设备算法实现。下面我们以逐系数操作为例来说明。之所以选择这个例子，是因为逐系数操作在不同设备上的实现逻辑相似，这样便于对比分析。cwise_ops_common.h 文件中定义的类模板 `BinaryOp` 是逐系数运算中的二元操作符抽象，它的签名为 `template <typename Device, typename Functor> class BinaryOp : public BinaryOpShared`，其中 `Device`、`Functor` 分别是设备和操作符的类型参数。在 `BinaryOp` 类的计算方法 `Compute` 中，`Device` 参数作为函数模板 `OpKernelContext::eigen_device` 的类型参数传入。在 op_kernel.cc 文件中，该函数模板针对 CPU、CUDA GPU 和 OpenCL GPU 设备分别做了特化实现，返回 Eigen 库中定义的不同设备类型的指针。`Compute` 方法的计算过程实现依赖

cwise_ops.h 文件中定义的结构模板 `BinaryFunctor`。该结构是二元操作符的执行器，签名为 `template <typename Device, typename Functor, int NDIMS, bool has_errors = Functor::has_errors> struct BinaryFunctor`。`BinaryFunctor` 结构针对三类设备分别做了偏特化实现，以设备特有的方式调用 `Functor` 参数实例化之后对应的二元操作。因此，模型参数完全实例化之后的 `BinaryOp` 对象可以在其 `Compute` 方法中访问特定类型的设备，并以设备特有的方式执行计算逻辑。对于已在 cuBLAS、cuDNN 等库中定义过的算法，它们对应的操作执行器结构的 CUDA GPU 偏特化实现直接调用这些库中定义的函数，以便获得更好的性能。

12.4.2　系统软件接口

TensorFlow 支持对接的系统软件包含两类：本地操作系统以及以文件系统为代表的第三方平台层软件或服务（PaaS）。系统软件接口相关的代码位于源代码包的 tensorflow/core/platform 目录，图 12-10 给出了系统软件接口相关的主要数据结构及其继承关系。

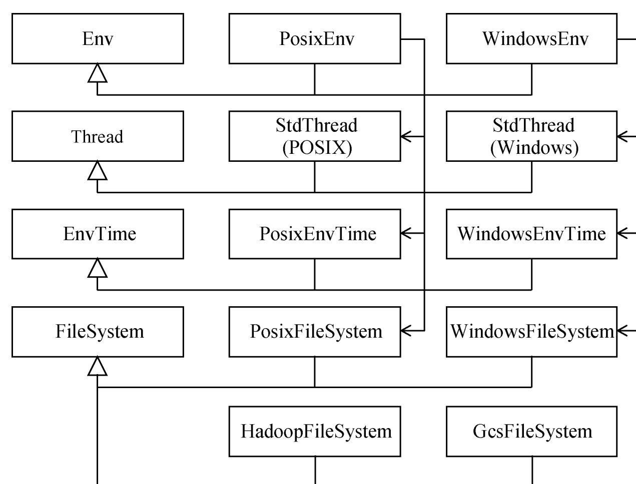


图 12-10　系统软件接口相关的主要数据结构及其继承关系

在本地操作系统方面，TensorFlow 支持实现了 POSIX 标准的类 Unix 系统（包括 Linux、macOS 等）以及 Windows 系统。TensorFlow 提供一组公共接口类型，作为访问不同操作系统共性能力的基类。其中，`Env` 类以统一的接口集成了本地操作系统的多类系统调用，涉及文件系统操作、线程管理、计时等，它是 TensorFlow 核心访问本地操作系统功能的主要入口。`FileSystem`、`Thread`、`EnvTime` 等基类则是对操作系统特定组件的抽象，便于不同的本地操作系统接口子类模块化地实现相应功能。面向类 Unix 系统和 Windows 系统的接口实现子类分别位于 posix 和 windows 子目录，这些子目录下另提供操作系统专有的动态链接库加载和网络端口检查等方法。

在第三方平台层软件或服务方面，TensorFlow 以编译时可选特性方式提供对 HDFS 分布式文件系统和 Google 云服务的接入能力。对于 HDFS 及 Google 云服务中的 GCS（Google Cloud Storage），

接口子类 HadoopFileSystem 和 GcsFileSystem 继承了同本地文件系统相同的 FileSystem 类，因此 TensorFlow 核心能够以一致的接口透明地访问异构存储。为了在 Google 公有云上运行 TensorFlow 实例，云服务接口模块还提供了用户认证、HTTP 请求管理、时间戳解析等功能。

下面以文件系统为例，说明系统软件接口的设计与实现原理。文件系统接口基类 FileSystem 提供创建、删除、重命名文件和目录的虚方法。辅助类型 RandomAccessFile 和 WritableFile 作为随机访问文件和可写文件的抽象基类，提供文件读、写、追加、同步等虚方法。辅助类型 ReadOnlyMemoryRegion 是内存映射文件的抽象基类，提供访问内存数据的虚方法。每种本地或远程文件系统的接入代码需要继承并实现这些抽象类型。例如，HDFS 文件系统的接入代码通过封装 libhdfs 库，将基类方法映射到 HDFS 的具体实现。上层代码使用 Env 或其他基类的方法访问文件，不必关心文件操作的具体实现。TensorFlow 对文件系统的接入支持涉及编译时和运行时两个层面。编译时支持体现在 configure 选项，以及 Bazel 构建脚本基于操作系统类型的选择性编译。运行时支持体现在基于 URL 前缀的文件系统自动选择机制。

为了实现异构文件系统的统一管理及自动选择机制，TensorFlow 提供文件系统注册工具类 FileSystemRegistry 及其实现类 FileSystemRegistryImpl。FileSystemRegistry 类的 Register 方法及其封装宏 REGISTER_FILE_SYSTEM 用于将一种文件系统接口子类注册到 TensorFlow 运行时环境，并将一个作为 URL 前缀的协议名字符串与之关联（对于本地文件系统，协议名可以是空串）。Lookup 方法通过协议名查找并返回对应文件系统的接口子类对象。GetRegisteredFileSystemSchemes 方法用于列出所有已注册的协议名。因此，当上层代码访问形如 hdfs://localhost:8020/path/to/file 的文件名时，文件系统接口能够自动将请求转发到 HDFS。

此外，TensorFlow 还提供文件系统插件机制。该机制允许在不重新编译 TensorFlow 源代码的前提下，在应用程序运行过程中加载第三方提供的动态链接库，实现对其他文件系统的访问。以 Python API 为例，文件系统插件加载函数为 tensorflow.load_file_system_library。由此也可以看出，TensorFlow 的系统软件接口设计较为开放，使用方式非常灵活。

12.5　小结

TensorFlow 的运行时核心是一套使用 C++ 语言开发的动态链接库，实现了平台层业务逻辑。运行时核心能够接入多种异构计算设备，并为应用模型开发者提供设备无关的多语言编程抽象和 API。运行时核心的各个模块遵循层次化设计，其中公共运行时是数据流图执行逻辑的核心层次，分布式运行时是实现平台可伸缩性的基础。运行时核心包含若干关键数据结构、公共基础模块和外部环境接口，它们属于 TensorFlow 的全局共性机制，为各个功能模块所共享。XLA 编译优化技术的引入，使得 TensorFlow 核心代码在加速器和终端硬件上具有更加高效的运行时形态。单元测试框架的引入，为平台层开发者学习 TensorFlow 核心机制的使用提供了典型的样例代码。理解 TensorFlow 运行时库的整体框架和共性机制，是进一步分析具体功能模块设计原理的基础。

第 13 章

通信原理与实现

13

TensorFlow 作为以数据流图的执行为主要业务逻辑、支持单机跨设备和多机分布式执行的计算平台，需要使用通信机制协助数据流图执行过程的运转。通信逻辑在 TensorFlow 系统设计中的地位较为关键，它既作为跨设备的张量数据传输机制，又成为调度流程的重要组成部分，还是分布式运行框架得以实现的基础技术。因此，我们有必要对 TensorFlow 的通信原理与实现进行专门的分析。TensorFlow 单机和分布式运行模式涉及的通信技术有所差异。本章依据两种场景对通信需求的不同，将通信机制分为进程内通信与进程间通信两部分加以解析。此外，RDMA 作为一种能够充分利用 InfiniBand 等高性能网络带宽的通信协议，也得到了 TensorFlow 的支持。本章亦会对这一模块进行介绍。

13.1 概述

TensorFlow 的通信机制主要服务于运行时核心组件间的数据交换和流程控制。在数据交换方面，通信的内容主要是数据流图上节点之间传输的张量，通信的源和目标主体一般是本地进程管理的不同设备，或是参与分布式计算的不同进程。在流程控制方面，通信的内容主要是与会话和数据流图具体执行逻辑相关的方法参数，通信的源和目标主体一般是进程内或进程间负责不同业务逻辑的对象。进程内通信技术主要实现内存数据的跨设备复制，进程间通信技术主要实现数据流和控制流基于网络协议的传输。

通信机制是 TensorFlow 应用程序实现可伸缩性（scalability）的基础，它使得同一套应用代码可以通过增加计算资源的方式提高计算性能。在使用 TensorFlow API 开发应用的过程中，通信接口对用户并不直接可见，因此属于一类“隐式”操作。TensorFlow 的运行时框架会依据应用代码中的集群与设备配置，在数据流图构建时自动插入通信操作，并在数据流图执行过程中动态调度执行这些操作。为了直观地说明通信机制在 TensorFlow 中的使用场景，图 13-1 给出了基于 PS-worker 模型的多机、多 GPU 分布式运行模式下，TensorFlow 所涉及的进程内和进程间通信关系示例。可以看出，进程内通信主要用于本地 CPU 与 GPU 内存之间，以及多个 GPU 内存之间的内存复制；进程间通信主要用于跨进程的数据传输，其中涉及 GPU 内存的数据传输需要经过 CPU 内存中转，除非使用支持 RDMA 协议的高性能网络且启用了第三方贡献的 GDR（GPUDirect RDMA）特性。

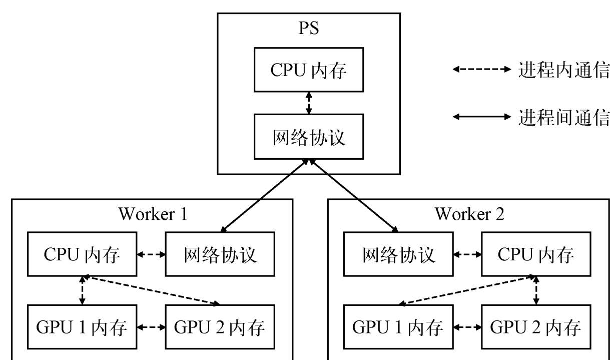


图 13-1 PS-worker 模型涉及的通信关系示例

TensorFlow 的通信机制设计遵循着追求高效的思想。例如，对于逻辑上具有数据传输语义，但物理上通信的源和目标位于同一设备、同一地址空间的进程内通信，只传输数据指针，不传输数据本身。再如，对于 CPU 和 GPU 内存之间的通信，尽量使用设备底层接口实现，而不使用面向应用层的 API。为了尽可能做到计算与通信过程的重叠，从而提升平台性能，TensorFlow 的很多通信函数采用异步模式及多线程并发执行。异步通信的结果通知机制——回调函数也是同 TensorFlow 调度框架总体设计相一致的机制，它使得通信过程能够无缝融入数据流图的整体执行过程。不过需要注意的是，对于同一台机器上多个进程之间的通信，除了操作系统内核固有的优化机制外，TensorFlow 开源版本暂未提供其他优化机制。这意味着以单机多进程方式使用 TensorFlow 时，即使多个进程使用了同一个设备，TensorFlow 也会使用基于网络传输的进程间通信机制实现内存复制。

13.2 进程内通信

对于单机运行模式以及分布式运行模式中每个进程内部的设备间数据交换需求，TensorFlow 提供一套基于会合点（rendezvous）的进程内通信机制。这套机制实现了一种以会合点为中心的生产者–消费者模型，并提供直观的消息传递（message passing）语义接口。它支持发送与接收操作的异步调用，并可以一致而透明地访问本地异构设备内存。进程内通信组件也是设计与实现进程间通信组件的基础。

13.2.1 通信接口

在 TensorFlow 中，无论进程内通信还是进程间通信，均使用统一的消息传递模型及一致的编程接口。在这种通信模型中，待传输的数据封装在消息抽象中，一次消息传递过程需要由发送者和接收者显式参与。对于最简单的一对一通信场景，每个发送操作均与唯一的接收操作配对。TensorFlow 的消息传递接口如同数据计算接口一样，被定义为数据流图上的操作。发送与接收的操作类分别是定义于 tensorflow/core/kernels/sendrecv_ops.h 文件中的 `SendOp` 与 `RecvOp` 类。发送

是一个同步操作，SendOp 类继承了 OpKernel 类并实现了 Compute 方法；接收是一个异步操作，RecvOp 类继承了 AsyncOpKernel 类并实现了 ComputeAsync 方法。传入 SendOp::Compute 和 RecvOp:: ComputeAsync 方法的 OpKernelContext 对象关联的输入或输出向量中，只有向量下标为 0 的唯一一个元素会得到处理，并作为发送的源或接收的目标。通信接口之所以能够以统一的入口方法处理进程内通信和进程间通信，是因为其方法参数传入的 OpKernelContext 对象内部可以用会合点基类指针指向不同类型的会合点子类对象，从而以多态方式调用不同场景下的通信实现代码。

不同于显式定义在应用代码中的操作，发送与接收操作一般是由 TensorFlow 核心隐式地加入到数据流图中的，这一逻辑是在图的提取和切分过程中实现的。在数据流图的待执行子图提取时，TensorFlow 需要为子图设置输入数据填充（feed）与输出数据获取（fetch）机制。如果没有启用基于 FunctionCallFrame 的数据传输优化路径，那么会合点将被用作数据填充与获取的中介。在这种情况下，tensorflow/core/graph/subgraph.cc 文件中定义的 subgraph::RewriteGraphForExecution 函数会重写子图的输入和输出节点，通过 FeedInputs 和 FetchOutputs 函数为图添加 RecvOp 和 SendOp，使得图的执行器能够通过会合点为子图填充输入数据，或从子图获得输出数据。图 13-2 给出了子图提取过程中添加通信操作的示例。

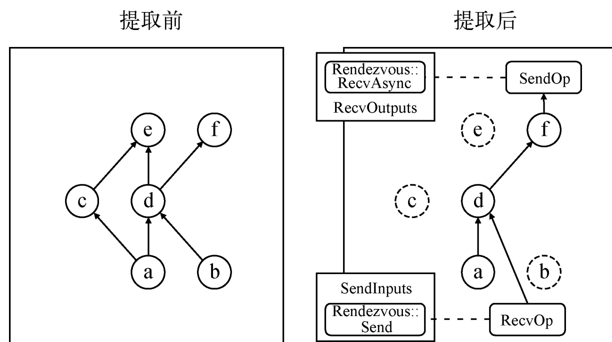


图 13-2 数据流图在子图提取过程中添加通信操作的示例

对于数据流图被切分到不同设备上执行的情况，TensorFlow 需要确保图切分前后具有相同的逻辑语义。数据流图上，一对数据收发操作恰好等价于一条切分前的边，因此可以通过插入通信操作节点来解决语义等价性问题。tensorflow/core/graph/graph_partition.cc 文件中定义的 Partition 函数能够消除跨设备的边，并通过 AddSend 和 AddRecv 函数在切分后的两个局部图上插入一对 SendOp 和 RecvOp 操作。Partition 函数还会检查同一个张量在源设备与目标设备间具有多条传输边的情况，对此加以优化，不添加重复的 SendOp 和 RecvOp 操作。图 13-3 给出了图切分过程中添加通信操作的示例。

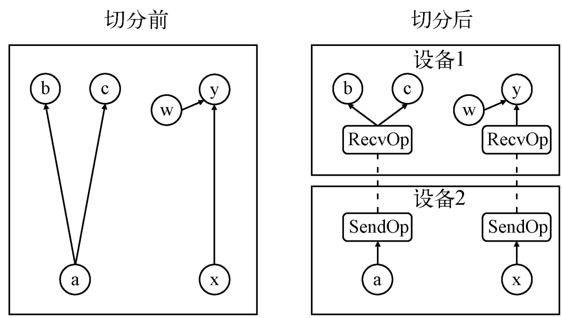


图 13-3　数据流图在按设备切分过程中添加通信操作的示例

13.2.2　会合点机制

会合点是分布式系统和消息传递模型中常用的一种机制，它用于协调异步执行的消息发送与接收操作，使得逻辑上关联的操作得以配对。在会合点缓冲区等机制的配合下，无论发送与接收操作调用的先后顺序如何、是否与其他同类操作穿插，均能够保证操作的正确配对和通信的顺利执行。TensorFlow 的消息传递操作也不例外，它以会合点作为发送与接收操作的中间协调者，以封装后的张量作为待传递的消息。发送操作不会因为接收操作尚未调用而阻塞，接收操作也不会因为发送操作尚未调用而出错。接收者总会在发送者完成张量传递之后得到通知，从而进行后续计算。会合点机制的相关数据结构定义在 tensorflow/core/framework/rendezvous.h 和 tensorflow/core/common_runtime/rendezvous_mgr.h 等文件中，图 13-4 给出了相关类的 UML 类图。

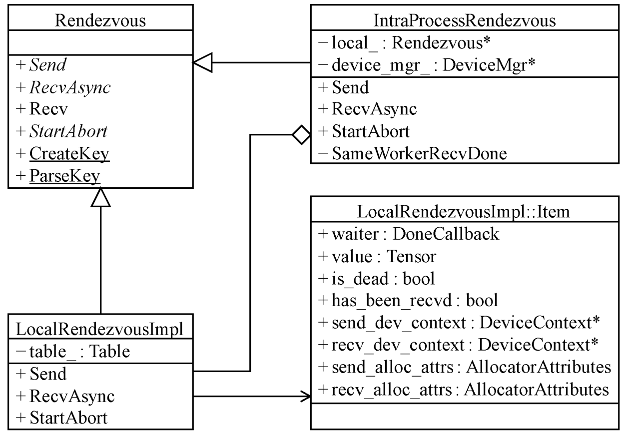


图 13-4　进程内会合点相关类的 UML 类图（只列出关键成员变量与方法）

`Rendezvous` 类是会合点的基类，它继承了 `RefCounted` 类，其对象可以通过引用计数管理生命周期。`Rendezvous` 类的目的在于为不同通信场景下使用的会合点子类提供统一的接口，这些接口包括以虚方法声明的同步发送方法 `Send`、异步接收方法 `RecvAsync`，以及用于取消操作的 `StartAbort` 方法。为了便于在某些场景下使用同步接收语义，`Rendezvous` 类基于 `RecvAsync`

封装了同步版本的 Recv 方法。为了标识张量并实现发送与接收操作的正确配对，TensorFlow 引入了键（key）的概念。键是一个字符串，它的格式为：

```
[src_device];[src_incarnation];[dst_device];[tensor_name];[frame_id]:[iter_id]
```

其中，src_device 和 dst_device 分别是源设备与目标设备的名称；src_incarnation 是源设备的具象化标识，用于识别设备是否被重新初始化过；tensor_name 是张量的名称；frame_id 和 iter_id 是与控制流管理结构 FrameAndIter 相关的参数。Rendezvous 类提供静态的 CreateKey 和 ParseKey 方法，用来创建和解析键；另提供 ParsedKey 结构，用于将键中的字段结构化保存。Send、RecvAsync 与 Recv 方法的核心参数是由 ParsedKey 和 Tensor 构成的键值对。ParsedKey 对象的引用用于标识张量，Tensor 对象的引用或指针作为指向待发送或待接收的张量内存地址。RecvAsync 方法还需要传入一个回调函数对象，它将在接收操作完成之后被调用。回调函数参数中包含接收到的 Tensor 对象的引用，以便接收方对其进行访问。

LocalRendezvousImpl 类是 Rendezvous 的子类，实现了同一本地设备上的会合点。该类内嵌名为 Item 的结构，用于封装张量以及与消息传递相关的属性。其中，函数对象类型的 waiter 成员用于保存 RecvAsync 方法传入的回调函数，布尔型变量 has_been_recvd 用于标识张量是否已经被接收。LocalRendezvousImpl 类还包含一个 Table（本质为 std::unordered_map<uint64, Item*>）类型的成员 table_，用于缓存该会合点上尚未配对的发送或接收操作。table_ 以张量收发操作键的 64 位散列值作为映射表项的键，以 Item 对象指针作为映射表项的值。Send 方法调用时，首先在 table_ 中查找待发送的键。如果键不存在，则说明待配对的 RecvAsync 方法尚未调用。此时创建一个 Item 对象，赋予其待发送的张量以及相关属性，并将其插入 table_。如果键存在，则说明 RecvAsync 方法已先于 Send 调用。此时将相应的 Item 对象标记为已接收，然后将 Send 待发送的张量作为参数，调用 RecvAsync 之前保存在 Item 对象中的回调函数。类似地，RecvAsync 方法调用时，也要在 table_ 中查找待接收的键。如果键不存在，则说明待配对的 Send 方法尚未调用。此时创建一个 Item 对象，赋予其回调函数对象以及其他属性，并将其插入 table_。如果键存在，则说明 Send 方法已先于 RecvAsync 调用。此时将相应的 Item 对象标记为已接收，然后将 Send 之前保存在 Item 对象中的张量作为参数，调用 RecvAsync 方法传入的回调函数。由此，LocalRendezvousImpl 实现了与收发操作顺序无关的本地消息传递过程。

IntraProcessRendezvous 类是 Rendezvous 的另一个子类，提供同一进程内跨设备通信的会合点功能。该会合点是 SendOp 与 RecvOp 操作的进程内通信能力实现者，它约定：发送操作始终访问本设备内存，接收操作可以访问本设备或其他设备内存。IntraProcessRendezvous 对象包含一个私有的 Rendezvous 指针，指向与之绑定的 LocalRendezvousImpl 对象，作为张量在同一本地设备内传输的实际执行者。对于发送操作，Send 方法直接调用内部 LocalRendezvousImpl 对象的 Send 方法，完成本设备内存上的张量发送。对于接收操作，RecvAsync 方法首先调用内部 LocalRendezvousImpl 对象的 RecvAsync 方法，完成同一设备内存上的张量接收。然后在传入 RecvAsync 的匿名回调函数中，调用私有的 SameWorkerRecvDone 方法。该方法依据张量最终接收设备的不同，选择在同一设备内存上复制缓冲区指针，或是通过 DMA 机制实现跨设备的内

存复制。上层调用者（`RecvOp::ComputeAsync`）传入 `RecvAsync` 方法的回调函数对象被逐级传入 `SameWorkerRecvDone` 或其内部的 `CopyTensor::ViaDMA` 方法，继而在张量到达最终接收设备之后得以调用。图 13-5 给出了进程内收发双方借助会合点进行通信的过程，图 13-6 进一步展示了张量传递涉及的主要函数调用流程，并解释了其中以 lambda 表达式方式定义的匿名函数对象的位置与功能。

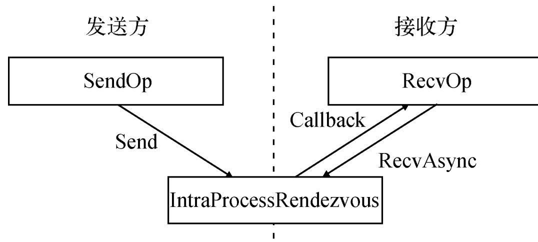


图 13-5　进程内收发双方借助会合点进行通信的过程

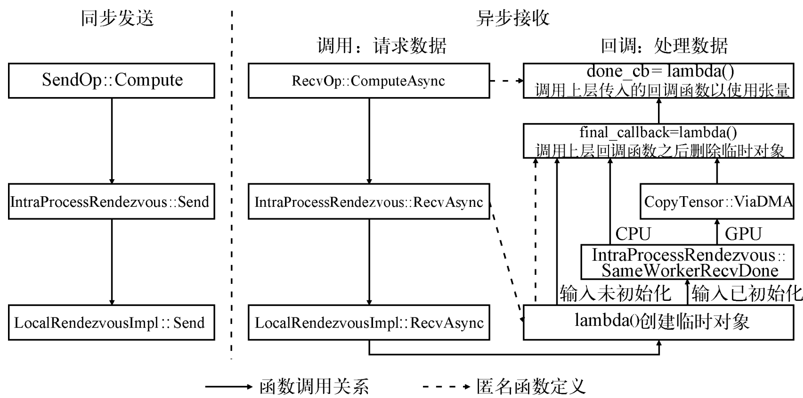


图 13-6　进程内会合点实现张量传递的函数调用与回调流程

13.2.3　异构设备内存访问

如 12.4.1 节所述，TensorFlow 目前的版本支持 CPU 设备，以及 CUDA 和 OpenCL（SYCL）这两类编程模型的 GPU 设备。进程内异构设备间通信的本质是跨设备内存的数据复制。现代 GPU 设备普遍支持直接内存访问（Direct Memory Access，DMA）能力，使得跨设备的内存复制过程无需 CPU 介入。然而，不同 GPU 编程模型的内存访问接口有所差异，获得通信性能的最佳实践也各有不同。TensorFlow 对异构设备内存访问的方法进行了封装，为会合点等上层调用者提供一致而高效的访问接口。对于跨设备内存间的张量复制，顶层工具类是定义在 tensorflow/core/common_runtime/copy_tensor.h 文件中的 `CopyTensor` 类，它的主要接口是静态的 `ViaDMA` 方法。该方法以张量名称、源和目标 `Tensor` 对象指针、双方所属设备的 `DeviceContext` 和 `Device` 对象指针，以及双方内存分配器所需的属性对象 `AllocatorAttributes` 为主要参数，并接受一个在复制

完成时调用的回调函数对象。该方法根据源设备和目标设备的类型，选择与设备匹配的内存复制函数执行复制操作。

在 TensorFlow 的实现中，`Tensor` 对象及内部 `TensorBuffer` 对象本身始终保存在主机（即 CPU）内存上，`TensorBuffer` 内部指针指向的缓冲区有可能位于 CPU 或 GPU 内存。张量的跨设备复制主要是指内部缓冲区的复制过程。CPU 内存之间的数据复制无须使用特殊的函数，直接调用 `Tensor` 类重载的赋值操作符即可。该赋值操作符内部通过 `Tensor::CopyFromInternal` 方法复制 `Tensor` 对象。对于 `TensorBuffer` 指向的缓冲区，不做完整复制，只需复制指针并增加引用计数。CPU 与 GPU 内存之间的数据复制函数是定义在 `DeviceContext` 类中的 `CopyCPUTensorToDevice` 和 `CopyDeviceTensorToCPU` 方法。不同 GPU 设备的 `DeviceContext` 子类需要重写这两个方法，以实现设备特有的缓冲区复制逻辑。同类或异类 GPU 设备内存之间的数据复制函数需要通过 `CopyTensor` 的内部类 `Registration` 以及私有方法 `Register` 注册到 `CopyTensor` 类中。目前，TensorFlow 只提供了 CUDA GPU 之间的内存复制函数，即 `GPUUtil::DeviceToDeviceCopy`。该函数实现的是缓冲区的完整复制。TensorFlow 已实现的异构设备间内存复制函数与设备的对应关系如图 13-7 所示。

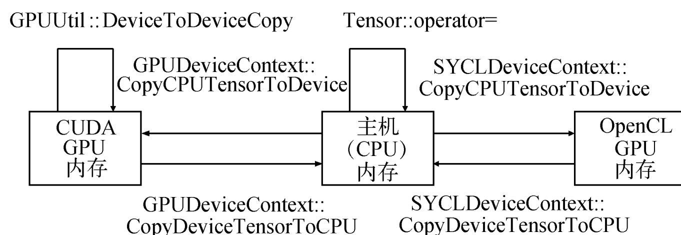


图 13-7 TensorFlow 已实现的异构设备间内存复制函数与设备的对应关系

对于 CUDA GPU 设备，包括内存复制逻辑在内的主要代码位于 tensorflow/core/common_runtime/gpu 目录。`GPUDeviceContext` 类重写的 `CopyCPUTensorToDevice` 和 `CopyDeviceTensorToCPU` 方法内部调用 `GPUUtil` 类的 `CopyCPUTensorToGPU` 和 `CopyGPUTensorToCPU` 方法实现内存复制。这些方法并没有简单地使用 CUDA API（如 `cudaMemcpy`），而是调用 `StreamExecutor` 提供的内存复制机制。`StreamExecutor` 直接基于 CUDA 驱动层接口实现，并提供异步复制和完成回调功能，在牺牲了一定易用性的前提下获得了执行性能方面的优势。

对于 OpenCL GPU 设备，包括内存复制逻辑在内的主要代码位于 tensorflow/core/common_runtime/sycl 目录。`SYCLDeviceContext` 类重写的 `CopyCPUTensorToDevice` 和 `CopyDeviceTensorToCPU` 方法并没有使用 OpenCL API（如 `clEnqueueCopyBuffer`），而是调用了 Eigen 项目封装的一套基于 SYCL 的内存复制方法。SYCL 提供的“单源”代码编写和编译模式有助于简化软件开发和构建流程。

13.3 进程间通信

进程内通信只满足了 TensorFlow 单机运行模式中的数据交换需求。对于分布式运行模式，

为了实现不同进程之间的协同工作，还需要进程间通信能力。TensorFlow 使用 Google 公司开源的远程过程调用库——gRPC 实现进程间通信。通过进程间的控制交互与数据交换，会话和数据流图上的操作得以在分布式集群上的多个进程内并发地执行。理解进程间通信的设计与实现是进一步理解 TensorFlow 分布式计算原理的基础。

13.3.1　gRPC通信机制

我们已在 2.2 节中提到过，gRPC 是一种客户端–服务器模式的开发库，它为开发者提供了一种接口调用方与接口实现方位置分离的函数调用机制。开发者只需编写扩展名为 .proto 的接口描述文件，gRPC 编译器即可自动创建目标语言的数据结构及其序列化方法，以及用于客户端、服务器端通信的接口代码。gRPC 的语义模拟了本地函数调用，每次调用（call）均由配对的请求（request）消息与响应（response）消息构成。客户端与服务器端可以通过有状态的通道（channel）自由连接，请求与响应消息的数据类型可以通过数据结构自由组合。因此，gRPC 通信机制能够为分布式应用开发提供较高的灵活性。

为了有效管理 gRPC 接口层的逻辑与抽象，同时易于上层代码使用进程间通信机制，TensorFlow 核心对 gRPC 库进行了封装，相关代码位于 tensorflow/core/distributed_runtime/rpc 目录内。尽管该目录中的文件名多以“grpc_”开头，然而它们并非 gRPC 库本身的代码副本，而是只包含 TensorFlow 的封装层代码。与 gRPC 通信相关的主要实体如图 13-8 所示。作为主要组件的 gRPC 服务器（server）被抽象为 `GrpcServer` 类，它提供 master 与 worker 两个服务（service），二者的接口定义于 tensorflow/core/protobuf 目录下的.proto 文件中。分布式模式中的每个进程具有一个 `GrpcServer` 实例，其中的 master 服务主要用于会话相关操作，worker 服务主要用于数据流图相关操作。表 13-1 列出了 master 与 worker 服务的主要接口及其功能说明。这些接口可分为控制通信接口与数据通信接口两类：大多数接口用于管理 TensorFlow 运行过程中会话和图的生命周期或执行逻辑，属于控制通信接口；worker 服务中的 `RecvTensor` 接口用于传输张量，属于数据通信接口。

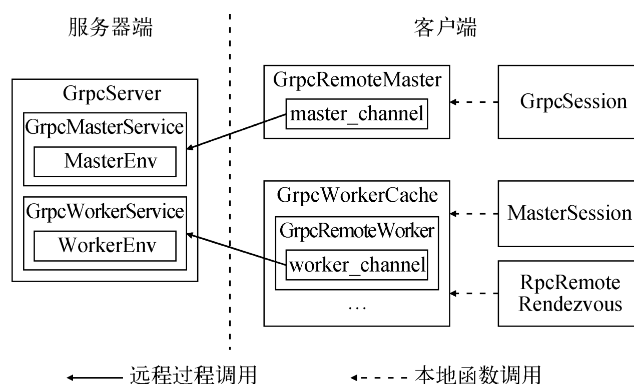


图 13-8　gRPC 通信相关的主要实体

表 13-1　GrpcServer 的主要接口功能说明

master 服务接口	功能说明	worker 服务接口	功能说明
CreateSession	创建会话及其数据流图	GetStatus	获取设备状态
ExtendSession	扩展会话的数据流图	CreateWorkerSession	创建 worker 会话对象
PartialRunSetup	设置运行部分图的参数	RegisterGraph	注册将要执行的数据流图
RunStep	驱动数据流图计算	DeregisterGraph	注销不再使用的数据流图
CloseSession	关闭会话	RunGraph	执行已注册的所有数据流图
ListDevices	列出 master 可用的设备	CleanupGraph	清理特定执行步的存留状态
Reset	关闭所有存在的会话	CleanupAll	清理所有存留状态
		RecvTensor	接收特定执行步中指定键所对应的张量
		Logging	日志（开源版本中暂未实现）
		Tracing	调试信息（开源版本中暂未实现）

master 与 worker 服务的服务器端与客户端实现分别定义于 GrpcMasterService—GrpcRemoteMaster 和 GrpcWorkerService—GrpcRemoteWorker 两组类中。GrpcMasterService 与 GrpcWorkerService 是服务器端接口的实现类。为了方便 gRPC 服务同 TensorFlow 的设备管理、线程调度等机制交互，两个服务分别包含了 MasterEnv 与 WorkerEnv 结构，其中封装了服务用到的多种组件指针。GrpcRemoteMaster 和 GrpcRemoteWorker 是客户端“存根”（stub）类，是远程服务在本地的代理，提供形同本地函数调用的接口，以便隐藏远程调用的通信逻辑。代表客户端与服务器端连接关系的实体是通道（grpc::Channel），TensorFlow 提供与每个服务独立关联的通道指针（类型定义为 SharedGrpcChannelPtr），以及用于记录通道元信息的 GrpcChannelSpec 类。客户端存根对象构造时，以通道指针作为连接指定服务器端的参数。master 服务的主要调用者是 GrpcSession 类，worker 服务控制通信接口的主要调用者是 MasterSession 类，数据通信接口的主要调用者是 RpcRemoteRendezvous 类。

进程间执行远程过程调用的前提是启动 gRPC 的服务器端与客户端，并建立两者之间的连接。对于分布式运行模式，每个进程同时具备 gRPC 服务器端与客户端两种角色。应用启动时，每个进程均对应一个“主机名:端口”参数，该参数来自应用层 API 的 ClusterSpec 对象，保存在核心层的 GrpcChannelSpec 对象中。这个参数指定当前进程的 GrpcServer 实例作为服务器端所开放的网络端口，也是其他进程作为客户端连接该进程 gRPC 服务的依据。在服务器端方面，GrpcServer 实例初始化时，将新建的 master 与 worker 服务关联到该端口，并启动监听。两个服务的消息处理循环函数（HandleRPCsLoop）在各自独立的线程中并发执行，异步处理来自客户端的请求。在客户端方面，针对 master 服务，指向目标会话所在进程的通道及 GrpcRemoteMaster 存根对象在本地的 GrpcSession 实例创建时建立。针对 worker 服务，TensorFlow 提供一套通道缓存机制，该机制实现于 GrpcWorkerCache 类。本地 GrpcServer 实例初始化时，向缓存中添加 GrpcChannelSpec 对象保存的所有进程的通道。当任何对象需要调用指定进程的 worker 服务时，可以查找缓存的通道指针，并基于它创建 GrpcRemoteWorker 存根对象。

gRPC 的服务器端与客户端均支持同步与异步两种实现模式。同步模式的客户端函数调用需要阻塞等待服务器端执行完毕才能返回；同步模式的服务器端只有完成一个调用请求才能接受下一个请求。异步模式的客户端在函数调用之后立即返回，客户端随后可以通过完成队列取回已完成调用的结果；异步模式的服务器端能够并发接受多个服务调用请求，以多线程方式在后台执行，并通过完成队列获知某个调用已经处理完成。在 TensorFlow 的两个 gRPC 服务中，为提升并发处理性能，服务器端均以异步模式实现。客户端则依接口的需求不同，有同步和异步两种实现模式。其中，master 服务的客户端调用多为同步模式，worker 服务的客户端调用多为异步模式。

需要注意的是，TensorFlow 的通信机制包含若干本地优化路径的设计。即对于调用同一进程内的 master 或 worker 服务的情况，均有可能绕过 gRPC 接口，直接访问本地实现对象。特别对于 master 服务，由于当前 TensorFlow 上层 API 设计的局限性，客户端或选择使用本地优化路径，或将通道指向本地 gRPC 服务，其通信往往退化为进程内通信。但由于 master 和 worker 服务本质上是为分布式平台服务的，这里将其统一作为进程间通信机制讨论。事实上，即使在进程内通信场景下，这些 gRPC 服务依然可以为运行时库提供异步调度能力。TensorFlow 未来版本或许会提供 master 服务器与客户端分离的通信模式，从而实现应用代码与运行时代码解耦的分布式框架。

在研读基于 gRPC 开发的通信代码时，如果发现一些类型或方法明显带有 TensorFlow 语义，却无法在 TensorFlow 源代码中找到，那么它们对应的源文件很可能是在 TensorFlow 构建时由 Protocol Buffers 编译器基于 .proto 文件生成的。专注于 TensorFlow 技术的读者一般不需要深入研究 Protocol Buffers 及其 gRPC 插件所创建的晦涩的内部实现代码，只需要了解这些自动生成的类型或方法的惯用命名规则和接口规范即可。例如，接口响应消息的数据类型是“`[RPC 函数名]+Response`”，数据结构内部成员的赋值方法是“`set_[成员名]`”等。此外，由于 TensorFlow 和 gRPC 均为 Google 公司开发，出于执行效率考虑，TensorFlow 使用了若干 gRPC 未提供文档的内部函数。这些函数的语义可根据其名称推测，一般不影响 TensorFlow 层面的代码理解。

13.3.2 控制通信

master 与 worker 服务提供的多数接口用于控制通信。下面分析控制通信服务器端与客户端的工作流程与原理。

1. 服务器端逻辑

服务器端方面，要理解控制通信的设计，首先需要了解 gRPC 的基本抽象：异步服务（`AsyncService`）、完成队列（`CompletionQueue`）、标记（`tag`），以及 TensorFlow 对于 gRPC 调用在服务器端的生命周期管理类——`Call`。

Protocol Buffers 编译器生成服务代码时，会为每个服务的异步版本创建名为 `grpc::[服务名]::AsyncService` 的类。启动和管理服务时，可以使用 `grpc::ServerBuilder` 类。`ServerBuilder` 对象启动异步服务的一般步骤是顺序执行以下方法：新增监听端口（`AddListeningPort`）、注册服务（`RegisterService`）、新增并返回完成队列（`AddCompletionQueue`）、构建并启动服务

（BuildAndStart）。对于 TensorFlow，这些步骤均在 GrpcServer::Init 方法或其级联调用的方法中完成。

gRPC 的完成队列是一种针对异步事件的管理机制，其功能是将期望发生的事件加入队列，在事件发生之后由队列的 Next 方法返回一个标记。出于为开发者提供灵活性的考虑，gRPC 将标记定义为 void 指针。开发者可以通过自定义类型扩展标记的语义，从而通过标记访问发生事件的对象。服务器端的完成队列类型为 grpc::ServerCompletionQueue。通过调用 AsyncService 类中自动生成的名为 Request+[RPC 函数名]的方法或者通用的 RequestAsyncUnary 方法，服务器端可以将期望处理的远程过程调用加入到完成队列中。完成队列的 Next 方法阻塞地等待并返回队列中与已加入调用相关的事件标记，这些事件可能包括请求收到、处理完成等。每次调用 Request+[RPC 函数名] 或 RequestAsyncUnary 方法时，只代表期望收到相应 RPC 函数的一次调用请求。当该请求被处理完成后，服务器端不再处理同一 RPC 函数的后续调用，除非将其再次加入队列。异步服务的后台处理是并发的，Next 方法返回标记的顺序由事件时序决定，与向完成队列加入调用的顺序无关。在 TensorFlow 中，异步服务类 grpc::MasterService::AsyncService 和 grpc::WorkerService::AsyncService 的对象及其对应的完成队列分别封装于 GrpcMasterService 和 GrpcWorkerService 类中。两个服务各自的消息处理循环函数 HandleRPCsLoop 轮询检测完成队列 Next 方法返回的标记，以便处理相关事件。HandleRPCsLoop 函数在 GrpcServer::Start 方法中以新建线程的方式启动。GrpcServer 的 Init 与 Start 方法均在进程启动之初、GrpcServer 对象创建后随即执行。

TensorFlow 基于 gRPC 的异步事件管理机制提供了自己的调用类与标记类，并实现了一套基于状态转移与回调函数的服务器端调用处理流程。相关数据结构的定义位于 grpc_call.h 文件中。图 13-9 给出了服务器端异步状态管理相关类的 UML 类图。其中，Call 类是 gRPC 调用在服务器端的抽象。它继承自 UntypedCall 类，在此基础上使用 C++ 模板参数化方式关联到特定的｛服务，请求消息类型，响应消息类型｝元组，从而能够代表特定 RPC 函数的调用。Call 类还包含指向对应服务接口实现函数的指针。UntypedCall::Tag 类是一种带有函数回调功能的标记抽象，它的对象指针作为加入完成队列的标记。其关键方法是 OnCompleted，该方法由消息处理循环函数在标记从完成队列返回之后调用。Tag 对象具有一个回调函数枚举参数（Callback），用来指示调用 OnCompleted 时所执行的操作。每个 Call 对象包含 3 个 Tag 对象：request_received_tag_、response_sent_tag_ 和 cancelled_tag_，它们分别关联到请求收到、响应发出和请求取消这 3 种事件。

服务器端处理一次调用的主要流程如下。

(1) 创建 Call 对象，将枚举参数为“请求收到”的 Tag 对象指针加入到服务的完成队列。这一过程实现于 Call 类的静态方法 EnqueueRequestForMethod，并通过 ENQUEUE_REQUEST 宏统一封装。每个服务接口的首次 ENQUEUE_REQUEST 操作位于服务的 HandleRPCsLoop 函数起始，其中对于那些需要频繁并发调用的接口，一次性入队 100~1000 个对象指针。以后，在每个调用处理完成时，执行一次 ENQUEUE_REQUEST，使得同一接口的下一次调用能够被处理。

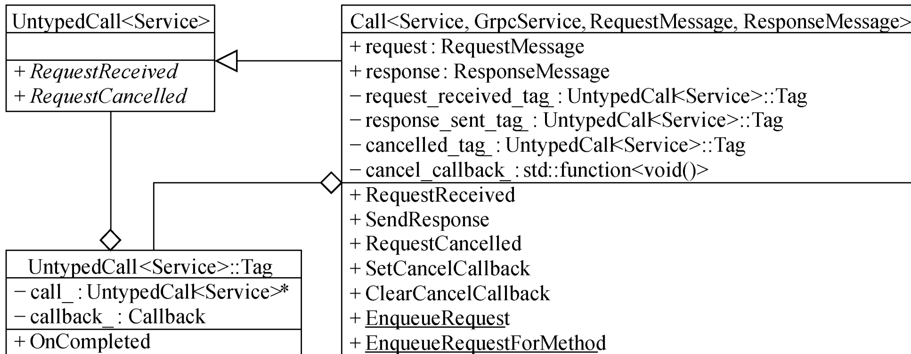


图 13-9　服务器端异步状态管理相关类的 UML 类图（只列出关键成员变量与方法）

(2) 收到客户端发来的调用请求后，`HandleRPCsLoop` 函数中完成队列的 `Next` 方法会返回一个枚举参数为“请求收到”的 `Tag` 对象，该对象的 `OnCompleted` 方法调用后，触发 `Call` 对象的 `RequestReceived` 方法。在该方法中，`GrpcMasterService` 或 `GrpcWorkerService` 中的接口实现函数（名为`[RPC 函数名]+Handler`）通过预设的指针得以调用。`Call` 对象的指针作为参数传递给接口实现函数，以便后者取得调用对应的请求消息，并为之生成响应消息。

(3) 接口实现函数处理请求消息的计算逻辑、生成响应消息之后，调用 `Call` 对象的 `SendResponse` 方法。该方法通过 gRPC 内部的 `grpc::ServerAsyncResponseWriter::Finish` 函数将响应的处理工作收尾，留待 gRPC 后台调度发送。`Finish` 函数会将枚举参数为“响应发出”的 `Tag` 对象加入到完成队列。

(4) 响应消息由 gRPC 后台发送给客户端之后，`HandleRPCsLoop` 函数中完成队列的 `Next` 方法会返回一个枚举参数为“响应发出”的 `Tag` 对象，该对象的 `OnCompleted` 方法调用后，触发 `Call` 对象将自身删除，从而完成一次 gRPC 响应的生命周期维护。

服务器端 gRPC 调用管理的一种特殊情况是请求因某些异常而取消。对于支持取消的 RPC 函数，其 `Call` 对象可以通过 `SetCancelCallback` 方法设置取消之后执行的回调函数。`Call` 类内部的 `RegisterCancellationHandler` 方法使用 gRPC 服务器上下文管理类 `grpc::ServerContext` 的 `AsyncNotifyWhenDone` 方法，将 `Call` 对象中枚举参数为“请求取消”的 `Tag` 对象指针加入到服务的完成队列。当请求正常结束或被取消时，完成队列的 `Next` 方法都会返回该 `Tag` 对象指针。这时 `Tag` 对象的 `OnCompleted` 方法将触发 `Call` 对象的 `RequestCancelled` 方法。该方法通过检查 `ServerContext` 对象的 `IsCancelled` 返回值，判断此事件是否由请求取消所致，若是则调用事先设置的回调函数，使得服务可以清理相关资源。

图 13-10 展示了服务器端异步调用的生命周期管理中 `Call` 对象的状态转换关系。

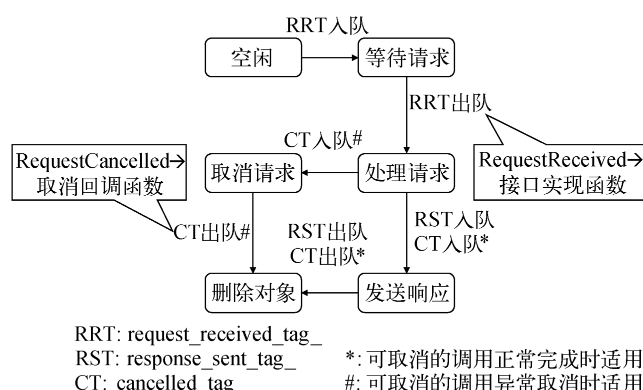


图 13-10　服务器端 `Call` 对象状态转换图

2. 客户端逻辑

在客户端方面，同步客户端的设计实现非常简单。对于以同步调用为主的 master 客户端，存根类 `GrpcRemoteMaster` 提供一组与 RPC 函数同名的代理方法，其内部使用 `grpc::BlockingUnaryCall` 函数阻塞调用 Protocol Buffers 编译器生成的底层存根对象上的方法，实现远程过程调用。

异步客户端的设计实现与异步服务器端有相似之处，均使用完成队列管理异步事件。客户端的完成队列类型为 `grpc::CompletionQueue`，其消息处理循环是在 `GrpcWorkerCache` 对象构造时，通过创建在新线程上执行的匿名函数实现的。该匿名函数轮询检测完成队列 `Next` 方法返回的标记，以便处理相关事件。对于以异步调用为主的 worker 客户端，存根类 `GrpcRemoteWorker` 提供一组名为 `[RPC 函数名]+Async` 的代理方法，这些代理方法通过 `IssueRequest` 函数启动一次异步请求的生命周期。图 13-11 给出了客户端异步状态管理相关类的 UML 类图。代表异步请求生命周期的数据结构是 `GrpcRemoteWorker::RPCState` 类，该类使用 C++ 模板参数化方式关联到特定的请求和响应消息类型。`RPCState` 类继承自标记类 `GrpcClientCQTag`，因此自身对象的指针即可作为完成队列中的标记使用。与服务器端的 `Tag` 类相似，`RPCState` 类也有名为 `OnCompleted` 的方法，由消息处理循环函数在标记从完成队列返回之后调用。`RPCState` 对象初始化时需要提供一个回调函数对象，该回调函数将在 `OnCompleted` 方法调用后执行。

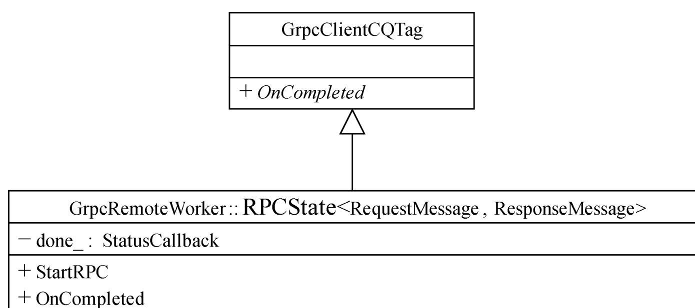


图 13-11　客户端异步状态管理相关类的 UML 类图（只列出关键成员变量与方法）

客户端发起一次异步调用请求并取得响应的主要流程如下。

(1) 上层代码调用 GrpcRemoteWorker 上的代理方法时，传入请求消息指针、待填充的响应消息指针，以及回调函数对象。代理方法将这些参数传递给 IssueRequest 函数。IssueRequest 函数生成 RPCState 对象，调用该对象的 StartRPC 方法。StartRPC 方法通过 gRPC 内部的 grpc::ClientAsyncResponseReader::Finish 函数将请求的处理工作收尾，留待 gRPC 后台调度发送。Finish 函数会将 RPCState 对象自身的指针作为标记加入到完成队列。

(2) 收到服务器端发来的响应后，消息处理循环函数中完成队列的 Next 方法会返回请求的标记，即 RPCState 对象指针。该对象的 OnCompleted 方法调用后，触发之前代理方法传入的回调函数，以便客户端得知调用完成，可以处理已经就绪的响应消息。随后，OnCompleted 方法将 RPCState 对象自身删除，从而完成一次 gRPC 请求的生命周期维护。

图 13-12 展示了客户端异步调用的生命周期管理中 RPCState 对象的状态转换关系。

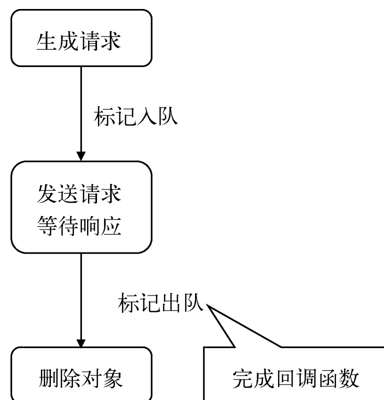


图 13-12　客户端 RPCState 对象状态转换图

13.3.3　数据通信

TensorFlow 核心的数据通信接口是 worker 服务的 RecvTensor 接口。从 gRPC 实现的角度而言，数据通信与控制通信并无本质区别，其差异只在于上层代码提供的消息数据结构、接口实现函数不同。控制通信的服务器端与客户端异步处理流程对于数据通信完全适用。然而，RecvTensor 接口又有一定的特殊性，主要体现在其响应消息内容是张量。张量的生命周期管理不仅与通信相关，还与计算逻辑相关；张量的大小依应用不同有较大的浮动范围，可以从几字节到数百兆字节。为了高效地实现数据通信，TensorFlow 设计了一些服务于 RecvTensor 接口的特殊机制。此外，进程间的张量传输也需要使用会合点机制实现通信双方的同步，这种会合点称为远程会合点。下面分别介绍 RecvTensor 接口的特殊设计以及远程会合点的实现原理。

RecvTensor 接口的第一个特殊机制是自定义的消息传输与解析方式。不同于仅返回简单响

应甚至空响应的控制通信接口，RecvTensor 返回的响应消息有可能很大。如果使用 gRPC 自动生成的序列化机制处理响应消息，将会产生较大的数据编码、解析与复制开销。为了避免这一开销，TensorFlow 使用字节缓冲区（grpc::ByteBuffer）及自定义的序列化、反序列化方法传输张量。考虑到接口的易用性，TensorFlow 并没有将字节缓冲区暴露在 RecvTensor 接口，而是隐藏在其 gRPC 封装层代码内部，并提供中间类型 TensorResponse 辅助上层调用。

在服务器端，RecvTensor 的接口实现函数 GrpcWorkerService::RecvTensorHandlerRaw 对应的 Call 类（WorkerCall<RecvTensorRequest, grpc::ByteBuffer>）通过模板参数化将响应类型定义为 ByteBuffer。在计算逻辑完成、待发送的张量生成之后，接口实现函数内部定义的完成回调函数将会调用定义在 grpc_tensor_coding.cc 文件中的 EncodeTensorToByteBuffer 或 EncodeRecvTensorResponseToByteBuffer 函数，将有可能保存在不同种类设备内存上的张量数据以尽可能减少内存分配与复制次数的方式写入 ByteBuffer 对象，供 gRPC 后台发送。写入 ByteBuffer 对象的字节流格式与 gRPC 自动生成的 RecvTensorResponse 类的序列化格式相同。

在客户端，RecvTensor 在存根类的代理方法 GrpcRemoteWorker::RecvTensorAsync 声明的响应类型为 TensorResponse。它之所以能够将服务器端以 ByteBuffer 类型发来的数据正确解析为 TensorResponse 类型，是因为 TensorFlow 使用 C++ 模板特征萃取（trait）方式，对 grpc::SerializationTraits 类在 TensorResponse 类型上的反序列化方法 Deserialize 进行了特化实现（定义于 grpc_worker_service_impl.h 文件）。该特化实现调用 TensorResponse 对象的 ParseFrom 方法，以尽可能高效的方式为本对象关联的 Tensor 对象填充数据。不同于仅存储张量及相关元信息的 RecvTensorResponse 类，TensorResponse 类还包含与设备管理、内存分配相关的对象指针字段，既能够将张量一次性反序列化到指定设备的内存，避免内存复制开销，又能够配合封装层的其他逻辑实现对张量生命周期的管理。这也是在客户端使用 TensorResponse 类取代 RecvTensorResponse 类的理由之一。

RecvTensor 接口的另一个特殊机制是客户端的调用生命周期管理。不同于仅在代理方法内部管理异步请求生命周期的控制通信接口，RecvTensor 另有一个封装在代理方法外部的客户端生命周期管理类——RpcRecvTensorCall。这个类定义于 rpc_rendezvous_mgr.cc 文件，它的主要目的在于：一是实现与 RecvTensor 调用相关的异构设备内存分配、TensorResponse 对象管理等辅助操作；二是建立 RecvTensor 与会合点机制的关联，使其能够驱动进程间数据通信。RpcRecvTensorCall 类的常用方法包括 Init 与 Start，分别用于初始化和启动 RecvTensor 调用。Init 方法传入 RecvTensorRequest 对象所需的执行步 ID 和会合点键，以及设备指针、内存分配器参数和操作完成回调函数，从而为本对象即将执行的 RecvTensor 调用提供必要的信息。Start 方法由私有的 StartRTCall 方法实现，它在指定设备上分配 TensorResponse 对象的内存，然后将 Init 方法传入的回调函数简单包装，传递给代理方法 RecvTensorAsync，从而启动 RecvTensor 调用。调用完成之后，Init 方法传入的回调函数可以通过 RpcRecvTensorCall 的 tensor 方法访问返回响应中的 Tensor 对象。为了避免并发的外部调用频繁创建和删除 RpcRecvTensorCall 对象的开销，TensorFlow 提供了一个可复用 RpcRecvTensorCall 对象的列表，

这些对象由 RpcRecvTensorFreeList 类管理。

进程间张量传输的会合点与进程内的设计有一定相似性，相关类的 UML 类图如图 13-13 所示。远程会合点实现于 RpcRemoteRendezvous 类，它逐级继承了名为 BaseRemoteRendezvous 和 RemoteRendezvous 的基类。基类存在的主要价值是为支持其他远程访问方式提供扩展能力。BaseRemoteRendezvous 对象包含一个私有的 Rendezvous 指针，指向与之绑定的 LocalRendezvousImpl 对象。后者的功能同其在进程内会合点场景下一致，是张量在同一本地设备内传输的实际执行者。BaseRemoteRendezvous 类约定：发送操作始终访问本设备内存，接收操作可以访问本地进程或远程进程内存。每个全局唯一的执行步对应到一个远程会合点对象。继承自 RendezvousMgrInterface 接口的 BaseRendezvousMgr 类及其子类 RpcRendezvousMgr 用于对本地的所有远程会合点对象进行管理，可以通过执行步 ID 查找或创建相应的会合点实例。

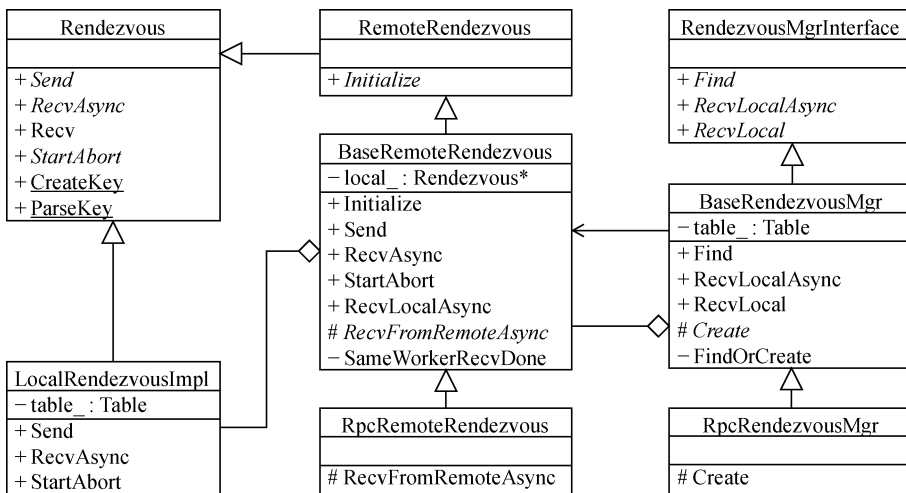


图 13-13　远程合点相关类的 UML 类图（只列出关键成员变量与方法）

远程会合点为 worker 服务器端提供的能力体现在 BaseRemoteRendezvous::RecvLocalAsync 方法。RpcRemoteRendezvous 继承了父类的这个方法，用于异步接收本地数据流图计算生成的张量，以便封装到 RecvTensor 的响应消息中。该方法借助 RecvLocalAsyncInternal 方法，将参数转发给内部 LocalRendezvousImpl 对象的 RecvAsync 方法处理。对于 RecvTensor 接口，其服务器端实现函数 GrpcWorkerService::RecvTensorHandlerRaw 在内部以异步线程方式调用 GrpcWorker::RecvTensorAsync 方法。这个方法可以通过 WorkerEnv 指针访问 RpcRendezvousMgr 对象，继而找到与当前执行步 ID 匹配的 RpcRemoteRendezvous 对象，然后调用该对象的 RecvLocalAsync 方法实现张量在本地的获取。

远程会合点为 worker 客户端提供的能力体现在 BaseRemoteRendezvous::RecvAsync 方法。RpcRemoteRendezvous 继承了父类的这个方法，用于选择性地从本地或远程获取张量。它会根据 RecvOp 传入的键进行判断：对于张量源和目标位于本地的情况，转发给 LocalRendezvousImpl

对象的 RecvAsync 方法处理，其中跨设备的处理方式与 IntraProcessRendezvous 类的做法相似；否则，转发给实现于子类 RpcRemoteRendezvous 中的 RecvFromRemoteAsync 方法。该方法创建 RpcRecvTensorCall 实例，实现对 RecvTensor 接口的远程过程调用。此外，BaseRemoteRendezvous 类还提供本地发送张量的 Send 方法。该方法在 RecvTensor 接口中并不使用，而用于 SendOp 调用。

图 13-14 给出了收发双方借助远程会合点进行进程间通信的过程。其中，接收方的调用流程遵守编号所示的时序，它通过普通函数调用发出请求，以匿名回调函数方式取得响应。发送方 SendOp 的执行时机取决于数据流图的计算进度，与是否收到 RecvTensor 请求无关，因此图中 Send 方法没有标注明确的时序。这幅图也反映了 TensorFlow 进程间数据通信的惯例：通信是由接收方主动发起、发送方被动参与的；RecvOp 触发跨进程的数据传输，SendOp 仅处理本地元数据。

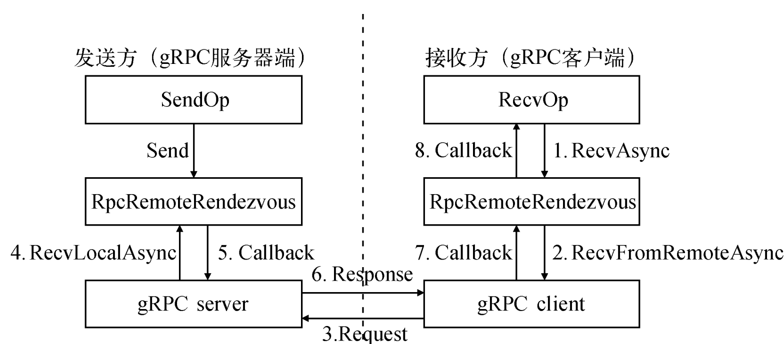


图 13-14　收发双方借助远程会合点进行进程间通信的过程

为了更清晰地说明 RecvTensor 接口所涉及的远程会合点操作与 gRPC 调用的执行逻辑，图 13-15 和图 13-16 分别给出了 RecvTensor 方法在服务器端和客户端的函数调用与回调流程，并解释了其中以 lambda 表达式方式定义的匿名函数的位置与功能。RecvTensor 相关的回调函数大多在其末尾嵌套调用上层传入的回调函数，因此它们的执行顺序恰好与其定义所在外层函数的执行顺序相反。利用捕获参数机制，回调函数一般会对其外层函数定义的变量进行后续处理，因此在语义上与外层函数具有密切的关联。可以看出，lambda 表达式机制在 TensorFlow 进程间数据通信中用得非常普遍，它为异步处理逻辑的实现提供了高度的灵活性。

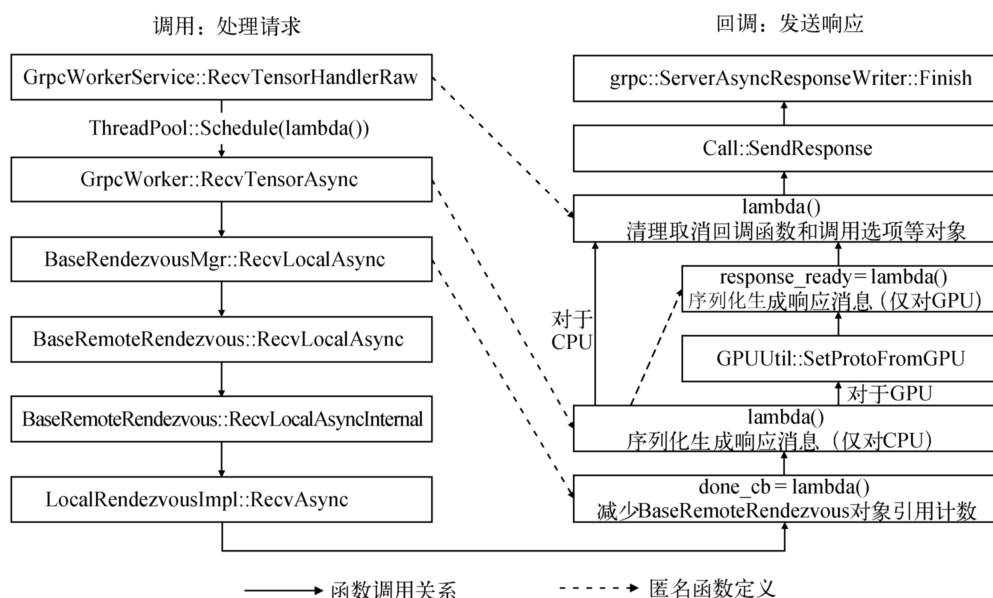


图 13-15　`RecvTensor` 方法在服务器端的函数调用与回调流程

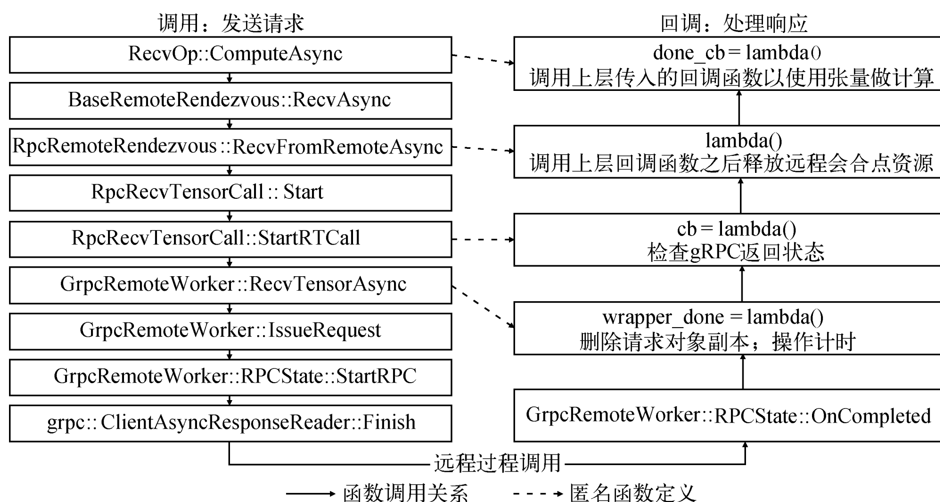


图 13-16　`RecvTensor` 方法在客户端的函数调用与回调流程

13.4　RDMA 通信模块

在商用集群中，1Gb/s~10Gb/s 的以太网是最常见的网络环境。但在高性能计算（High-Performance Computing，HPC）领域，InfiniBand 等高带宽、低延迟的高端网络才是主流的通信基础设施。随着互联网、大数据应用对通信能力要求的不断提高，这类普遍带宽达到 40Gb/s~300Gb/s 的高端

网络设备也逐渐走进了商用集群。InfiniBand 等高性能网络虽然兼容 TCP/IP 协议，但为了发挥性能极限，一般需要使用专用的网络协议。在 InfiniBand 协议栈中，远程直接内存访问（Remote Direct Memory Access，RDMA）特性得到了用户的重点关注和普遍使用。RDMA 特性能够减少通信过程中数据在不同层次内存之间的复制，降低了 CPU 参与通信时的指令、缓存与上下文切换开销，从而有利于提高吞吐率，缩短延迟，更好地实现计算与通信的重叠。

Google 公开资料显示，其公司内部的深度学习系统使用了 InfiniBand 网络与 RDMA 协议，不过 TensorFlow 的开源版本没有提供相关代码。好在开源社区能够群策群力，一些第三方贡献的组件增强了 TensorFlow 对高性能网络的支持。Yahoo! 发布的 RDMA 通信模块已被社区所接纳，放置在 TensorFlow 源代码包的 tensorflow/contrib/verbs 目录下。本节介绍这一模块的设计实现原理。

13.4.1　模块结构

RDMA 通信模块的基本设计思路是：保留 TensorFlow 原有的、基于 HTTP-TCP/IP 协议的 gRPC 进程间控制通信能力，将进程间数据通信的实现迁移到支持 RDMA 特性的 InfiniBand 协议栈（其通信原语称为 IB verbs）。在技术实现时，该模块利用了 TensorFlow 核心提供的若干种可扩展机制，基本做到了与原有代码的解耦。用户若需使用 RDMA 特性，只需要在编译 TensorFlow 源代码包之前，开启 configure 脚本中相应的配置项即可。

图 13-17 给出了 RDMA 通信模块的组件结构。为了保留原有的 gRPC 机制，该模块以继承自 `GrpcServer` 类的 `VerbsServer` 类作为框架性组件。`VerbsServer` 在继承 `GrpcServer` 原有 master 和 worker 服务的同时，额外提供 verbs 服务。这一 gRPC 服务用于传输 InfiniBand 通信原语所需的元信息，其唯一接口是 `GetRemoteAddress` 函数。verbs 服务的服务器端接口类和客户端存根类分别是 `GrpcVerbsService` 和 `GrpcVerbsClient`。在应用代码中，用户如果将 API 层 `Server` 对象（比如 Python API 中的 `tensorflow.train.Server`）的 `protocol` 属性设置为 `grpc+verbs`，TensorFlow 核心的服务器工厂机制就会创建 `VerbsServer` 对象以取代默认的 `GrpcServer` 对象，为 RDMA 通信提供服务。为了协调异步收发过程，RDMA 通信模块同样需要会合点机制。RDMA 通信的会合点类是 `RdmaRemoteRendezvous`，它是 `BaseRemoteRendezvous` 的子类，并具有名为 `RdmaRendezvousMgr` 的管理器类型。这一会合点的主要方法语义同 `RpcRemoteRendezvous` 相似。

图 13-17　RDMA 通信模块的组件结构

由于 IB verbs 的语义及调用规范与 gRPC 有较大差异，为了将 RDMA 通信操作高效地融入 TensorFlow 核心层的工作流程，RDMA 通信模块提供了一系列适配组件。`RdmaMgr` 是统筹管理 RDMA 通信相关对象的管理器组件，它为会合点对象提供了访问 RDMA 通信功能的接口。`RdmaAdapter` 组件封装了 InfiniBand 驱动及 IB verbs 接口，以便控制 InfiniBand 网卡完成 RDMA 通信逻辑。RDMA 通信模块将通信所使用的内存缓冲区（简称“RDMA 缓冲区”）抽象为 `RdmaBuffer` 类，将两个进程之间的通信链路抽象为 `RdmaChannel` 类。RDMA 通信链路上传输的消息由 `RdmaMessage` 结构封装，这些消息既包括用于管理通信时序的控制消息，也包括用于承载张量的数据消息。每个 `RdmaChannel` 对象包含一组 `RdmaBuffer` 对象的指针，它们分别指向一对控制消息收发缓冲区（message buffer RX/TX）、一对响应消息收发缓冲区（ACK buffer RX/TX）和一系列张量数据缓冲区（tensor buffer vector）。

13.4.2 消息语义

尽管 TensorFlow 的数据收发操作和 IB verbs 的通信原语都属于消息传递通信范畴，然而它们的接口语义以及对环境的要求有明显的不同。TensorFlow 的数据收发操作语义是与数据流图计算引擎的异步性、随机性相适应的。在对接 gRPC 通信库时，它能够利用 gRPC 函数调用通信接口内置的请求——响应匹配、消息内存动态分配等自动化机制，因此上层实现逻辑相对简单和直接。相比之下，IB verbs 的通信原语更加底层，不提供消息缓存、内存分配、动态调度等高级功能。它需要开发者显式维护消息收发所需的内存缓冲区，并要求发送方掌握代表接收方内存地址的远程内存键（remote key）。RDMA 通信还要求将缓冲区注册为锁页（pinned）内存，且内存注册过程存在较大开销。这些特点均与 TensorFlow 动态内存管理、异步操作调用的设计存在矛盾。

针对这些问题，TensorFlow 的 RDMA 通信模块提供了一套基于缓冲区预分配和消息交互的解决方案，该模块的工作流程很大程度上是围绕内存管理和时序控制展开的。

所谓缓冲区预分配，是指 RDMA 通信原语使用一组预先分配、相对静态的锁页内存缓冲区，而非 TensorFlow 固有的 `TensorBuffer`。在通信过程中，张量数据需要在两种缓冲区之间进行本地复制。虽然这有违 RDMA 开发中“零复制”的最优原则，但是它有效地权衡了运行时性能与易编程性。为了减少 RDMA 缓冲区分配和注册开销，同时提高内存利用率，RDMA 通信模块采用了一种基于数据流图语义的缓冲区复用机制。对于同一幅数据流图，同名张量进入下一执行步后，只要大小没有增长，其缓冲区就可以在执行步间复用。对于少数张量大小增长的特殊情况，可以实时分配新的 RDMA 缓冲区。

所谓消息交互，是指 RDMA 通信模块采用带有语义的消息实现通信时序管理。表 13-2 给出了该模块的消息类型及其功能简述。可以看出，除了用于张量传输的 `RDMA_MESSAGE_TENSOR_WRITE` 消息，其他消息均属于 TensorFlow 适配 IB verbs 通信模式所需的控制消息及其对应的响应消息。每种消息内部均包含各自的元信息字段，例如消息类型标识、执行步 ID、张量的名称和形状等。IB verbs 只支持传输字节流，不包含对数据结构进行序列化的功能。为了在网

络上传输结构化的消息，RdmaMessage 结构提供了用于数据序列化和反序列化的 CreateMessage 与 ParseMessage 方法。这对方法为不同类型的消息分别设计了优化的存储格式，它们以消息起始字节作为判断消息类型的依据。

表 13-2 RDMA 通信模块的消息类型及其功能简述

消息类型	功 能
RDMA_MESSAGE_ACK	响应已收到的控制消息
RDMA_MESSAGE_BUFFER_IDLE	声明本地张量数据缓冲区空闲可用
RDMA_MESSAGE_BUFFER_REQUEST	请求远程创建张量数据缓冲区
RDMA_MESSAGE_BUFFER_RESPONSE	反馈本地新创建的张量数据缓冲区
RDMA_MESSAGE_TENSOR_REQUEST	请求远程提供张量数据
RDMA_MESSAGE_TENSOR_WRITE	向远程写入张量数据

针对进程接收到的消息，RDMA 通信模块内置一套事件引擎机制。它在 RdmaAdapter 类的 Process_CQ 方法中实现了基于 InfiniBand 完成队列轮询检测的消息处理循环。Process_CQ 方法会在参与通信的进程中作为独立的线程运行，从而实时响应并异步处理其他进程发来的消息。表 13-3 给出了 RDMA 通信模块对不同类型消息的处理逻辑。

表 13-3 RDMA 通信模块的消息处理逻辑

消息类型	消息处理逻辑
RDMA_MESSAGE_ACK	标记远程控制消息缓冲区为空闲，发送下一条消息
RDMA_MESSAGE_BUFFER_IDLE	发送响应，标记远程张量数据缓冲区为空闲，发送下一条消息
RDMA_MESSAGE_BUFFER_REQUEST	发送响应，查找或创建张量数据缓冲区，发送 RDMA_MESSAGE_BUFFER_RESPONSE 消息
RDMA_MESSAGE_BUFFER_RESPONSE	发送响应，设置远程张量数据缓冲区信息，标记本地和远程缓冲区为空闲，发送下一条消息
RDMA_MESSAGE_TENSOR_REQUEST	发送响应，查找或创建张量数据缓冲区，发送下一条消息
RDMA_MESSAGE_TENSOR_WRITE	运行 RdmaRemoteRendezvous::RecvFromRemoteAsync 方法定义的回调函数

RdmaMessage 结构和序列化消息存储的元信息大多与 TensorFlow 核心层的语义相关。除此之外，RDMA 通信还需要 IB verbs 层面的通信接口元信息，例如 InfiniBand 通道的本地标识符（LID）、队列对序号（QPN）、包序列号（PSN），以及注册内存的地址和远程内存键等。这些元信息封装在 verbs_service.proto 文件定义的 Channel 和 MemoryRegion 数据结构中，通过 verbs gRPC 服务的 GetRemoteAddress 接口以“带外”方式传输。这一设计简化了 RDMA 通信的自举过程。

13.4.3 通信流程

TensorFlow RDMA 通信模块的工作流程遵循一条与原有 gRPC 通信机制相同的惯例，即 SendOp 只负责将待发送的张量加入本地会合点的消息列表，不主动进行数据传输；RecvOp 真正

触发通信过程，随后的数据传输由收发双方的远程会合点对象协作完成。RDMA 通信模块使用 13.4.2 节所述的消息组合取代 gRPC 通信机制中的 `RecvTensor` 接口。为简单起见，后面对通信参与方的称呼以张量传输过程为基准，将 `RDMA_MESSAGE_TENSOR_WRITE` 消息的收发双方定义为通信过程的“接收方”与“发送方”。

RDMA 通信模块触发数据传输的入口函数与 gRPC 机制的 worker 客户端入口流程类似，即经由 `RecvOp::ComputeAsync`—`BaseRemoteRendezvous::RecvAsync`—`RdmaRemoteRendezvous::RecvFromRemoteAsync` 方法启动张量接收。我们首先考虑最简单的情况，即收发双方用于存储张量的 RDMA 缓冲区已经就绪。图 13-18 给出了这种情况下双方的执行时序。

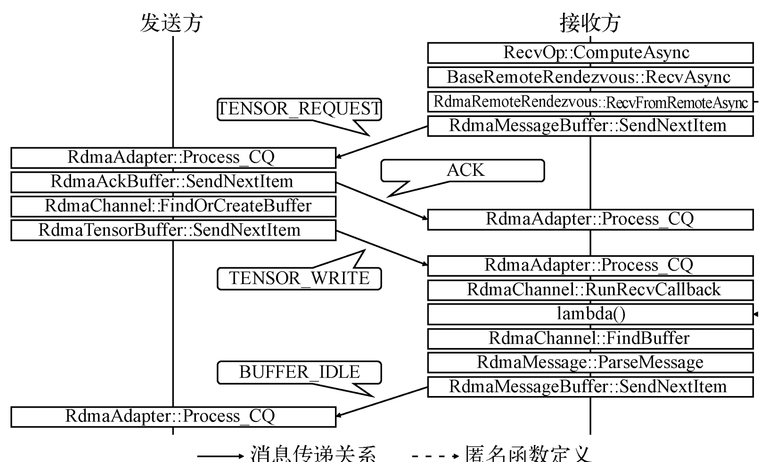


图 13-18　使用 RDMA 通信模块进行张量传输的典型执行时序

`RecvOp` 执行时，接收方的 `RdmaRemoteRendezvous::RecvFromRemoteAsync` 方法会调用 `RdmaMessageBuffer::SendNextItem` 方法，向发送方发出带有张量名称和执行步 ID 字段的 `RDMA_MESSAGE_TENSOR_REQUEST` 消息。发送方的 `RdmaAdapter::Process_CQ` 方法检测到这一消息后，首先发出 `RDMA_MESSAGE_ACK` 消息进行响应，然后找到待发送张量所在的缓冲区，并把用于发送张量的 `RdmaTensorBuffer::SendNextItem` 方法调度到线程池中执行。待该方法执行后，接收方的 `Process_CQ` 方法会检测到带有张量数据和元信息的 `RDMA_MESSAGE_TENSOR_WRITE` 消息。`Process_CQ` 方法对这种消息的处理流程是将本进程的 `RecvFromRemoteAsync` 方法事先注册的回调函数加载到线程池中执行。回调函数会从本地缓冲区中取出 `RDMA_MESSAGE_TENSOR_WRITE` 消息，将其字节流反序列化为 `Tensor` 对象，然后把对象复制到会合点参数指示的目标设备内存。处理完毕之后，回调函数随即向发送方发送 `RDMA_MESSAGE_BUFFER_IDLE` 消息，告知对方同一 RDMA 缓冲区已可为下一执行步提供服务。最后，这一回调函数还会继续调用上层传入的回调函数，以便操作节点使用张量数据执行计算逻辑。

接下来讨论现有张量数据缓冲区大小不能满足通信需求的情况，为此我们有必要理解张量发送方法 `RdmaTensorBuffer::SendNextItem` 的实现原理。该方法内部调用 `BaseRemoteRendezvous::`

RecvLocalAsync 方法，从会合点取得本地 SendOp 传入的数据。它将一个匿名的完成回调函数传入 RecvLocalAsync 方法，以便在会合点数据就绪时发送张量。这个回调函数会检查发送方缓冲区是否能够容纳待发送的数据。因为不同进程中的同名张量数据缓冲区在同一执行步中具有相同的大小，所以发送方的缓冲区大小也反映了接收方的缓冲区大小。如果缓冲区空间足够，那么回调函数会创建并发送前文所述的 RDMA_MESSAGE_TENSOR_WRITE 消息。否则，回调函数首先调用 RdmaBuffer::CreateCPUBuffer 方法，在本地创建更大的新缓冲区；然后创建并发送 RDMA_MESSAGE_BUFFER_REQUEST 消息，请求接收方也创建相同大小的新缓冲区。当发送方收到来自接收方反馈的 RDMA_MESSAGE_BUFFER_RESPONSE 消息后，消息处理函数会通过 RdmaChannel::FindBuffer 方法查找到此前创建的新缓冲区，随后再次调用 RdmaTensorBuffer:: SendNextItem 方法发送数据。图 13-19 展示了这一流程的执行时序。

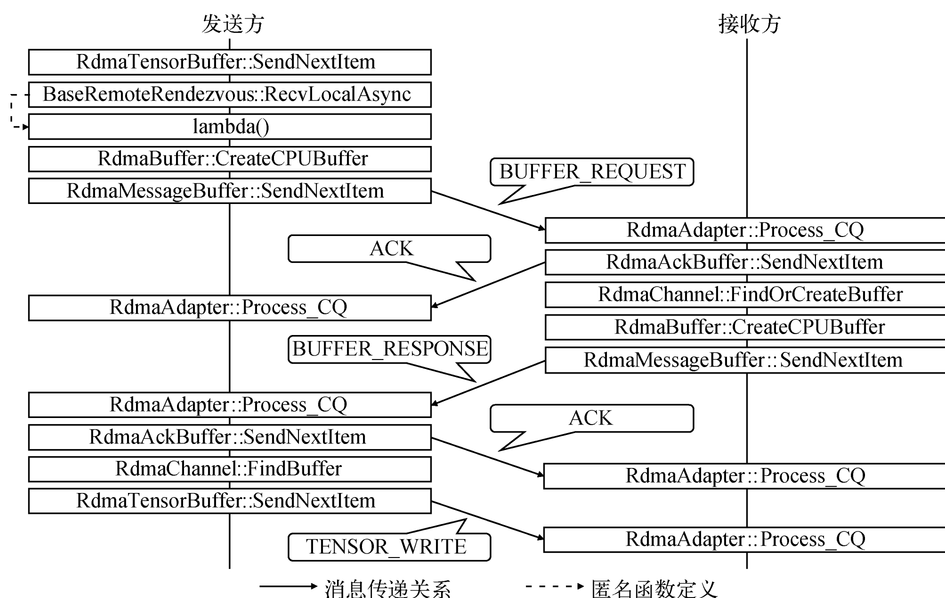


图 13-19 使用 RDMA 通信模块请求创建张量缓冲区的执行时序

最后，我们来看 TensorFlow RDMA 通信模块的消息收发操作如何落实于 InfiniBand 网卡。各个 RdmaBuffer 子类的 SendNextItem 方法最终都会调用 RdmaBuffer::Write 方法，该方法使用 IB verbs 的 ibv_post_send 原语，发起异步的远程内存写入操作。传统上，IB verbs 提供两类消息通信模式：双边收发（two-sided send/recv）与单边读写（one-sided read/write）。前者易于实施时序控制，后者具有更好的性能。RdmaBuffer::Write 方法没有选用这两种经典模式，而是采用另一种兼顾二者优势的折中模式：带有立即数据的 RDMA 写入（IBV_WR_RDMA_WRITE_WITH_IMM）。这种模式既能够以直接内存访问方式在远程设备中写入数据，有助于通信性能的提升；又允许接收方通过完成队列机制感知到通信的结束，有利于时序控制的简化。

消息处理循环的核心机制——InfiniBand 完成队列轮询检测是由 RdmaAdapter::Process_CQ 方法内部调用的 ibv_poll_cq 原语实现的，而消息接收则是由 Process_CQ 级联的 RdmaChannel::Recv 方法内部调用的 ibv_post_recv 原语实现的。这些 IB verbs 原语只会处理 InfiniBand 网卡的收发队列和完成队列相关事项，不会像以太网的 socket 接口那样在不同层次的缓冲区之间复制数据。这是因为，在 ibv_poll_cq 原语返回的前一刻，数据已经被远程进程写入了本地进程事先注册的锁页内存——这正体现了 RDMA 协议的能力与优势。有兴趣的读者可以阅读 InfiniBand 官方技术文档或 Linux 系统联机手册，了解有关 IB verbs 的更多知识。

13.5　小结

通信机制是 TensorFlow 平台的重要组成部分，也是数据流图执行过程不可缺少的环节。通信机制的存在，使得单机多设备及多机分布式的并行计算模式成为可能，并为跨设备和跨进程的运行时调度提供支持。TensorFlow 的通信机制包括进程内通信与进程间通信，二者具有统一的接口——SendOp 与 RecvOp 类，以及一致的异步收发协调机制——会合点。进程内通信组件基于 CUDA 驱动层接口和 SYCL 技术实现了数据在本地设备之间的传输。进程间通信组件基于 gRPC 库实现了数据流和控制流在分布式进程间的传输。RDMA 通信模块作为一种高性能的进程间通信组件，为 TensorFlow 充分利用 InfiniBand 等高端网络的能力提供支持。TensorFlow 通信机制的设计与实现以高效性为指导原则。总体上，通信接口基于异步模式设计，有助于实现计算与通信逻辑的并发执行。在 GPU 内存复制和 gRPC 数据序列化等局部技术点上，TensorFlow 设计了多种优化方案，尽可能减少不需要的开销以提升通信性能。理解通信机制是深入理解 TensorFlow 数据流图计算原理，尤其是分布式会话机制的基础。

第 14 章

数据流图计算原理与实现

TensorFlow 将神经网络训练与推理过程表达为数据流图计算，并使用会话抽象维护计算流程的生命周期与上下文。数据流图计算是 TensorFlow 运行时库最重要的工作职责，是将直观的应用层 API 语义映射到高效的核心层执行逻辑的精髓所在。读者学习数据流图计算的设计与实现，是理解 TensorFlow 核心原理的关键步骤，也是对运行时库进行二次开发的必备环节。本章首先介绍 TensorFlow 核心层的计算框架整体设计以及数据流图创建涉及的主要流程与抽象；然后针对单机与分布式两种运行模式，分析两种会话的设计原理及其典型的数据流图执行逻辑；最后简述图上操作节点在计算设备上的调度执行过程。

14.1 概述

TensorFlow 的数据流图计算过程，是将用户基于应用层 API 编写的、以神经网络算法为代表的计算逻辑转换为 C++ 核心层的细粒度、抽象化运行时形态，进而在计算设备上以一致而有序的方式调度执行的过程。核心层以较少的接口抽象对应用层 API 屏蔽了内部实现的复杂性。数据流图计算可认为包含控制流与数据流两条主线。在控制流方面，会话（session）是计算流程和相关对象生命周期的管理者，旨在保证计算时序的正确性与高效性。在数据流方面，图（graph）是算法逻辑和张量数据的承载者，图上元素的分解和重组实现了应用层数据结构同并行计算设备的桥接。

图 14-1 给出了数据流图计算的整体调用栈。我们可以将数据流图计算粗略地划分为应用程序逻辑、会话生命周期和算法核函数执行这 3 个层次。

- 在应用程序逻辑中，用户使用 Python 等应用层 API 及高层抽象编写算法模型，无须关心图切分、进程间通信等底层实现逻辑。算法涉及的计算逻辑和输入数据绑定到图抽象中，计算迭代控制语义体现在会话运行前后的控制代码上。
- 在会话生命周期层次，单机会话与分布式会话具有不同的设计：单机会话采用相对简单的会话层次与图封装结构，它将图切分、优化之后，把操作节点和张量数据提交给底层执行器；分布式会话分为 client、master 和 worker 三层组件，它们对计算任务进行分解和分发，并通过添加通信操作来确保计算逻辑的完整性。

- 在算法核函数执行层次，执行器抽象将会话传入的核函数加载到各个计算设备上有序执行。为充分利用多核硬件的并发计算能力，这一层次提供线程池调度机制；为实现众多并发操作的异步执行和分布式协同，这一层次引入了通信会合点机制。

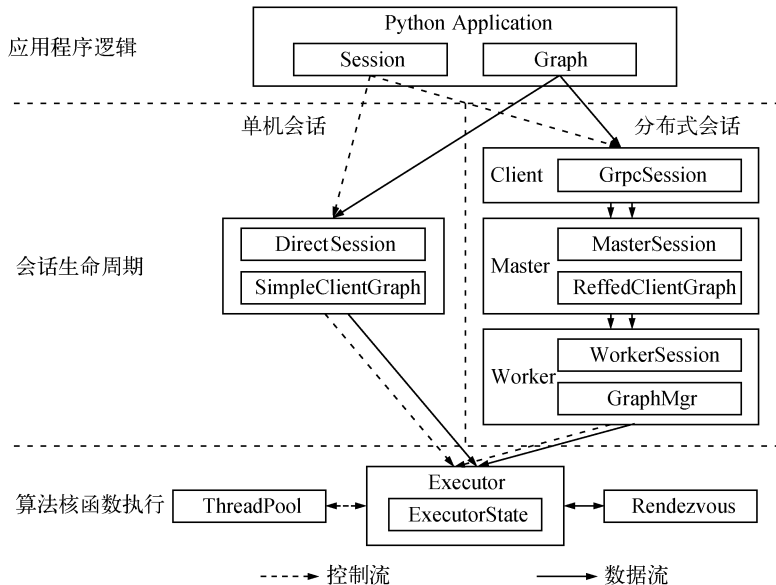


图 14-1　数据流图计算的整体调用栈

TensorFlow 数据流图计算机制的设计权衡了灵活性与高效性。在灵活性方面，TensorFlow 尽可能将不同层次的组件解耦，允许其在多种应用场景下复用。例如，执行器抽象同会话类型和设备种类无关，在不同的运行模式下能够以统一的接口操作核函数。在高效性方面，TensorFlow 针对特定场景设计了多种优化方法和旁路（bypass）路径，在不损坏系统整体逻辑的前提下增强局部执行性能。例如，master 组件具有用于本进程内访问的优化版本，能够避免不必要的网络传输开销。

14.2　数据流图创建

数据流图是 TensorFlow 计算过程的基本抽象。用户在程序代码中通过应用层 API 定义的数据流图并不能直接用于运行时核心的内部计算。它们将随着会话的构造和运行，被转换为运行时核心可执行的数据流图格式——这就是 TensorFlow 核心的数据流图创建过程。该过程的输入一般是应用层 API 定义的、直观表达计算逻辑的数据流图格式（例如 Python API 中的 `tensorflow.Graph` 类），输出为 TensorFlow 核心定义的、绑定到具体计算设备和算法实现的数据流图格式（即 C++ 代码中的 `Graph` 类及其成员类型）。在数据流图创建的过程中，基于 Protocol Buffers 文件定义的 `GraphDef` 等数据结构因具有跨语言、易操作的优势，常被一些中间函数用于表示拓扑、无计算

语义的数据流图中介格式。

创建数据流图的入口函数和上层调用路径依据应用使用的编程语言、API 版本及运行模式的不同而有所差异，但底层实现逻辑基本一致。本节以 TensorFlow Python API 开发的单机应用为例，说明数据流图创建的实现原理。

14.2.1　流程与抽象

数据流图创建过程主要包括全图构造、子图提取、图切分和图优化等阶段。为了说明 TensorFlow 核心层创建数据流图的过程，图 14-2 给出了一组在真实情况基础上简化过的典型流程示例，展示了一个简单的计算逻辑所创建的数据流图在各个阶段的变化。其中，节点内的字母代表其对应的操作或变量，节点背景颜色代表其被分配的计算设备。基于这一直观认识，我们进一步分析数据流图创建过程的内部实现。

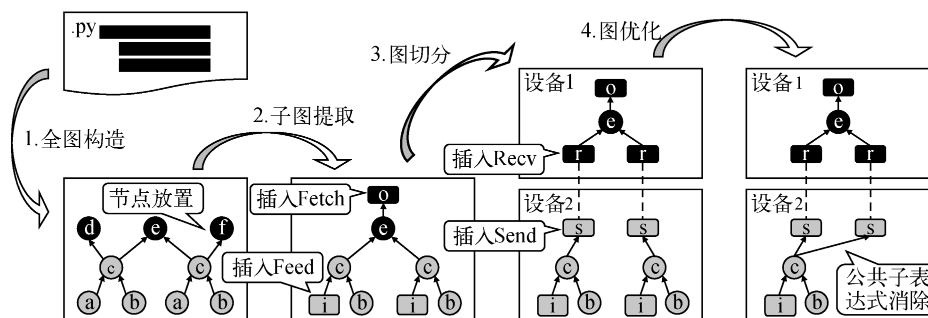


图 14-2　数据流图创建的典型流程

对于经由 Python API 调用会话接口的情况，TensorFlow 核心层的数据流图创建过程一般是在会话运行时启动的。`tensorflow.Session` 类的 `run` 方法及其级联调用的内部方法会依次调用 C API 的 `TF_ExtendGraph` 和 `TF_Run` 函数。前者实现会话及其绑定数据流图的初次构造或后续扩展，后者实现面向特定变量及其对应子图的会话运行。之所以需要在每次 `TF_Run` 之前均执行一次 `TF_ExtendGraph`，是为了让应用代码中的数据流图变化及时反映到核心可执行的数据流图中。在单机运行模式下，C API 的这两个方法内部分别调用 C++ 核心层 `DirectSession` 类的 `Extend` 和 `Run` 方法实现。图 14-3 和图 14-4 分别给出了 `DirectSession` 类在会话构造和运行时，涉及数据流图创建过程的主要流程。在图 14-3 和图 14-4 中，自上而下级联的方框表示主要方法及其调用关系，方框左上与右侧的类型名称表示特定方法中，与数据流图相关的输入、输出数据结构。

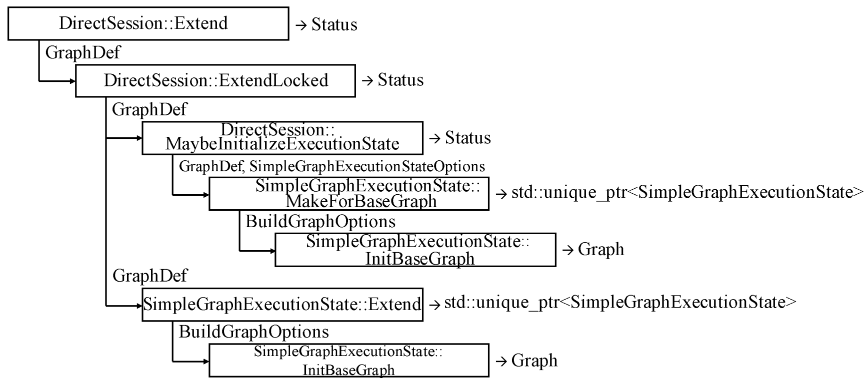


图 14-3　会话构造时涉及数据流图创建的主要流程

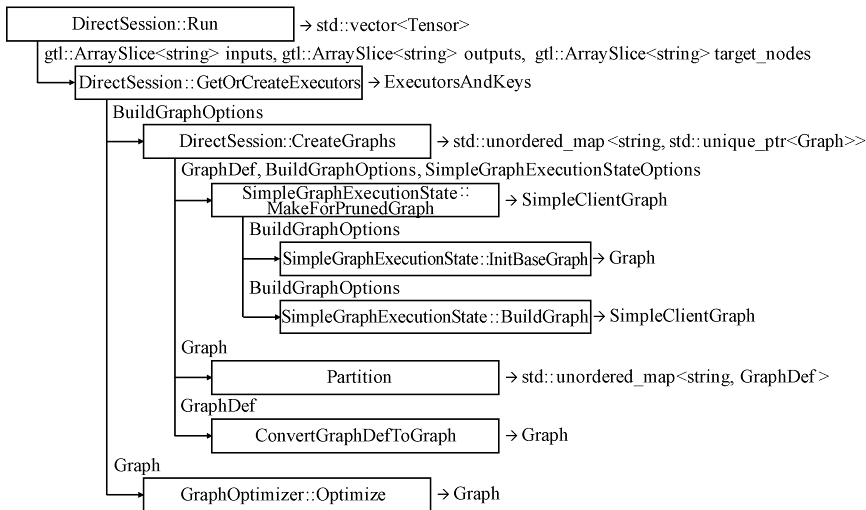


图 14-4　会话运行时涉及数据流图创建的主要流程

数据流图创建过程涉及的主要抽象是 `SimpleClientGraph` 结构及其状态管理相关类型，图 14-5 给出了它们的 UML 类图。`SimpleClientGraph` 结构定义在 tensorflow/core/common_runtime/simple_graph_execution_state.h 文件中，它封装了与会话的一次运行相关联的子图及其自定义函数库。会话运行时仅保存并处理必要的子图，能够降低内存占用和通信开销。`BuildGraphOptions` 结构定义了子图的输入、输出张量和目标节点，以及优化和调试选项，用作提取子图的参数。`SimpleGraphExecutionState` 类记录了数据流图节点在设备上的放置状态，它提供按需创建 `SimpleClientGraph` 实例的方法。其辅助数据结构 `SimpleGraphExecutionStateOptions` 用于保存设备列表和节点——设备映射关系。tensorflow/core/graph/graph_partition.h 文件定义的 `PartitionOptions` 结构则用于保存图切分过程的选项，它管理着图切分逻辑所需的外部函数指针，并提供控制流、计时器等元素的开关。

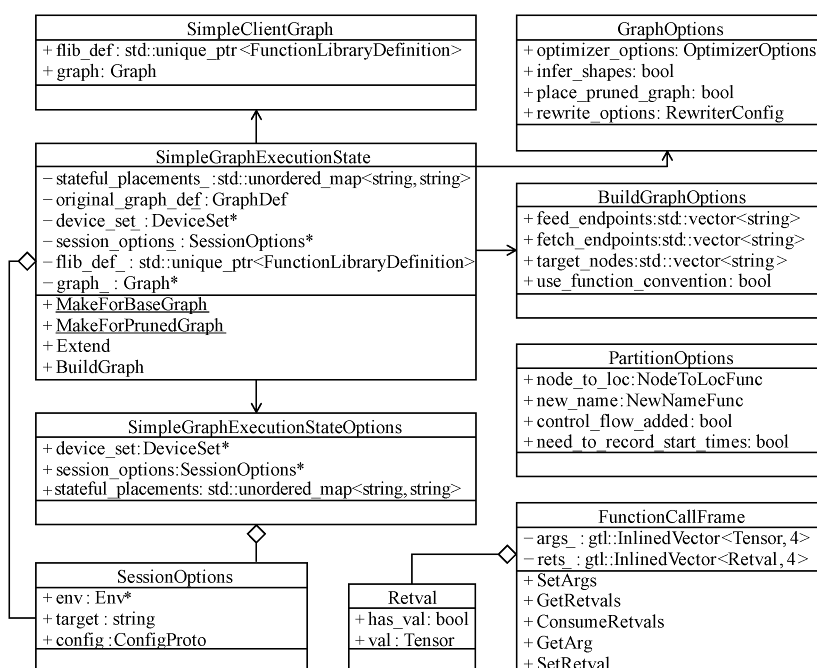


图 14-5　数据流图创建过程相关抽象的 UML 类图（只列出关键成员变量与方法）

在数据流图创建的过程中，作为会话级全局参数使用的数据结构包括 `SessionOptions` 和 `GraphOptions` 等。

- `SessionOptions` 结构保存着应用层 API 传入的会话构造参数，其中 `ConfigProto` 成员定义于 tensorflow/core/protobuf/config.proto 文件，它包括若干与数据流图创建、优化和运行过程相关的选项类型。
- `GraphOptions` 类是 `ConfigProto` 类的成员，用于在会话整体层面配置数据流图的属性。其主要成员包括子图提取、形状推导等特性的开关，图优化器、图重写逻辑等子过程的选项等。

子图提取过程还会涉及 TensorFlow 的函数调用上下文管理抽象——`FunctionCallFrame`。该类模拟了编程语言运行时的函数调用栈帧（frame in call stack）机制，是一种在用户代码中可访问的函数调用上下文。它包含基于 `Tensor` 数据类型的函数输入参数、返回值字段及其访问接口，并将返回值封装于辅助结构 `Retval` 中。相比编程语言原生的函数调用栈帧，`FunctionCallFrame` 抽象的主要优势在于其作用域和生命周期的可控性。

14.2.2　全图构造

全图构造过程在会话初次构造或后续扩展时执行，其目的是基于应用代码中逐次传入的、仅

包含基本拓扑信息的 GraphDef 格式数据流图，构造同会话生命周期关联的、具备计算能力的 Graph 格式的完整数据流图。

会话初次构造或后续扩展时，应用代码中定义的数据流图以 GraphDef 格式，经由 C API 传递到 DirectSession 对象的 Extend 方法，继而进入对图加锁保护的 ExtendLocked 方法。对于会话初次构造、图状态对象尚未初始化的情况，ExtendLocked 方法首先调用 MaybeInitialize-ExecutionState 方法，创建并初始化会话持有的 SimpleGraphExecutionState 成员，在该成员中保存会话关联的数据流图和设备等信息。MaybeInitializeExecutionState 方法内部构造 SimpleGraphExecutionState 对象并创建初始的 Graph 对象是通过调用 SimpleGraphExecutionState 类的 MakeForBaseGraph 静态方法实现的。该静态方法内部调用新状态对象的 InitBaseGraph 方法，实现图的格式转换及节点向设备的放置（placement）。节点放置机制实现于 SimplePlacer 类，该类提供一种基于并查集联合-查找（union-find）算法的简单节点放置策略，并具有 GPU 优先的启发式规则。

对于图状态对象已经初始化、会话中的数据流图需要扩展的情况，ExtendLocked 方法会调用 SimpleGraphExecutionState 成员的 Extend 方法，实现基于新加入数据流图的状态转移，并将新的状态对象赋予会话。SimpleGraphExecutionState::Extend 方法实现状态转移的主要步骤包括：

(1) 构建新的临时 GraphDef 对象；

(2) 将原图中的节点加入临时图；

(3) 基于 Protocol Buffers 的 MergeFrom 机制，将新图中的节点加入临时图；

(4) 计算生成临时图的 VersionDef 字段；

(5) 将新图的自定义函数库等字段赋予临时图；

(6) 基于临时图，构造新的 SimpleGraphExecutionState 对象，覆盖会话传入的原状态对象；

(7) 调用 SimpleGraphExecutionState::InitBaseGraph 方法，基于新状态对象中的 GraphDef 格式全图生成 Graph 格式全图，并完成节点向设备的放置。

14.2.3　子图提取

子图提取过程在会话运行时执行，其目的是构造关联到特定的一次会话运行过程、只包含特定计算任务相关节点的 Graph 格式子图。

会话运行时，DirectSession::Run 方法从 C API 得到本次运行涉及的输入张量集合以及输出和目标节点名称向量，然后将这些参数传入 GetOrCreateExecutors 方法。GetOrCreateExecutors 方法检查参数对应的、用于管理执行器的 ExecutorsAndKeys 对象是否已经存在，如果存在，则直接返回。否则，将这些参数封装到 BuildGraphOptions 结构，传递给 CreateGraphs 方法，

以便创建子图的 Graph 对象。读者可能已经注意到，CreateGraphs 方法名称中的 Graphs 一词为复数，其输出参数 outputs 是 Graph 对象指针的 std::unordered_map 容器。这是因为图切分过程也将由 CreateGraphs 方法调用，该方法最终返回的容器包含了已被切分到不同设备上的局部数据流图对应的 Graph 对象集合。

CreateGraphs 方法检查会话配置的 GraphOptions::place_pruned_graph 字段是否要求裁剪生成只包含待运行节点的子图。如果是，则调用 SimpleGraphExecutionState::MakeForPrunedGraph 静态方法。MakeForPrunedGraph 方法首先创建全图 SimpleGraphExecutionState 对象的副本，然后依次调用副本的 InitBaseGraph 和 BuildGraph 方法，生成裁剪过的子图，并将其封装到 SimpleClientGraph 对象，返回给会话。图的裁剪逻辑是在 SimpleGraphExecutionState::InitBaseGraph 方法内部调用的 subgraph::RewriteGraphForExecution 函数中实现的。该函数以 BuildGraphOptions 结构的成员为输入参数，在 Graph 对象上执行原地裁剪。如果会话配置不要求裁剪全图生成子图，那么 CreateGraphs 方法将直接调用全图 SimpleGraphExecutionState 对象的 BuildGraph 方法，生成对应的 SimpleClientGraph 对象并返回给会话。

为了确保子图可被独立执行，子图提取过程还需要为子图添加输入数据填充与输出数据获取操作。13.2.1 节已经介绍过一种可行的机制，即通过插入 SendOp 和 RecvOp，使用基于会合点的进程内通信实现数据传输。除此之外，TensorFlow 还提供一种基于 FunctionCallFrame 对象的数据传输优化机制。这种机制的开关是 BuildGraphOptions 结构的 use_function_convention 字段。当它开启时，subgraph::RewriteGraphForExecution 函数级联调用的 FeedInputs 和 FetchOutputs 函数会在子图中插入 ArgOp 和 RetvalOp，而非通信操作。这两种操作节点能够以更低的内存访问开销实现输入数据填充与输出数据获取。

14.2.4　图切分

图切分是指将一幅子图按照其操作节点放置的设备，切分为若干局部数据流图的过程。切分生成的每幅局部图仅在一个设备上运行，以便 TensorFlow 核心有效调度图上操作的执行。通信操作节点（SendOp、RecvOp）将被插入局部图，以确保执行子图的逻辑语义同切分之前一致。

图切分过程由定义在 tensorflow/core/graph/graph_partition.cc 文件中的 Partition 函数完成。该函数的输入参数包括 Graph 格式的子图及 PartitionOptions 类型的切分选项，输出参数是 GraphDef 格式的局部图集合。CreateGraphs 方法在完成子图提取之后，会构建一个 PartitionOptions 对象，并初始化其成员变量，然后调用 Partition 函数实施图切分。在 PartitionOptions 成员中，比较重要的是函数对象 node_to_loc 和 new_name。node_to_loc 指向的函数用于获取节点到设备的映射信息，当前的实现直接读取 Node 对象的 assigned_device_name 字段，该字段是由 SimplePlacer 类在图构造时赋值的。new_name 指向的函数用于生成局部图中的新节点名称，当前的实现使用原节点名称追加数字后缀的格式。

Partition 函数首先调用 AddControlFlow 函数，在数据流图中添加控制流。控制流用于确

保跨设备执行数据流图时，调度操作的时序正确性。然后，调用 `BuildMemoryDeviceInfo` 函数，在 `GraphInfo` 结构中记录图中每个节点的设备与内存类型信息，这些信息将作为实施图切分的参考依据。接下来，`Partition` 函数遍历图中所有节点，执行图切分算法。图切分算法会将每个节点加入对应设备的 `GraphDef` 格式局部图，并在图中重构必要的连接边。对于被切分的跨设备连接边，图切分算法会为之添加 `SendOp`、`RecvOp` 操作节点。`Partition` 函数执行完毕后，`CreateGraphs` 方法循环调用 `ConvertGraphDefToGraph` 函数，将切分后的局部图逐个转换为 `Graph` 格式。

14.2.5　图优化

图优化是指在数据流图运行之前，对图的节点和边进行适当调整，以便提升图运行效率的过程。TensorFlow 数据流图优化的技术原理与执行过程同计算机系统中其他常见的图优化技术（如编译器的语法树优化）类似。

`CreateGraphs` 方法返回切分过的局部数据流图集合后，`GetOrCreateExecutors` 方法会创建用于图优化的 `GraphOptimizer` 对象，然后对每幅局部图逐一执行 `GraphOptimizer::Optimize` 方法。该方法在图上依次执行多个优化函数，包括 `RemoveListArrayConverter`（删除列表和数组间的转换操作节点）、`RemoveDeadNodes`（删除具有无状态、不可达等特征的“死节点”）、`RemoveIdentityNodes`（删除与源和目标同时直接相连的 `Identity` 操作节点）、`ConstantFold`（常量折叠）、`OptimizeCSE`（公共子表达式消除）和 `ExpandInlineFunctions`（内联函数展开）等。这里简要介绍常量折叠和公共子表达式消除。

- 常量折叠是指对于只依赖于常量输入的计算节点，可以在图创建过程中直接计算结果，以包含结果的常量节点取代原有计算子图，从而避免图运行时反复执行同样的计算。常量折叠的实现位于 tensorflow/core/common_runtime/constant_folding.cc 文件，主要包含可折叠节点发现、常量图提取、常量图计算、节点替换等过程。
- 公共子表达式消除是指寻找图的拓扑中是否存在若干相同的子结构，当某种子结构使用两次以上时，后续使用的子结构无须重新计算，而可以直接引用第一次计算的结果。公共子表达式消除的实现位于 tensorflow/core/graph/optimizer_cse.cc 文件，其技术方案参考了编译器领域经典的全局值编号（Global Value Numbering）算法。

14.3　单机会话运行

TensorFlow 的核心业务逻辑是数据流图计算，承载这一逻辑的运行时机制是会话的运行。会话运行过程读入数据流图的待执行子图以及必要的输入张量，依据图中定义的依赖关系，将每个节点对应的操作核函数有序地加载到各个计算设备上并发执行，并将计算结果作为后续节点的输入或子图的输出。会话的生命周期最终完成子图上定义的所有计算语义，将输出结果以张量形式返回给创建会话的应用程序。本节介绍 TensorFlow 的单机运行模式下，会话运行机制的设计与实现。

14.3.1　流程与抽象

图 14-6 给出了单机运行模式下，作为核心抽象的 DirectSession 类在会话运行时的主要工作流程。会话运行的入口方法是 DirectSession::Run，其签名为 Status DirectSession::Run(const RunOptions& run_options,const NamedTensorList& inputs,const std::vector<string>& output_names,const std::vector<string>& target_nodes,std::vector<Tensor>* outputs, RunMetadata* run_metadata)。主要输入参数 inputs、output_names、target_nodes 分别是本次运行涉及的输入张量集合以及输出和目标节点名称向量。主要输出参数 outputs 是本次运行生成的张量集合。RunOptions 和 RunMetadata 参数是由 Protocol Buffers 文件定义的、通过应用代码传入的会话运行配置项和输出元信息。DirectSession::Run 内部包含执行器获取、输入数据填充（feed）、图运行、输出数据获取（fetch）和张量保存等 5 个主要步骤。其中，图运行是会话运行的核心步骤。

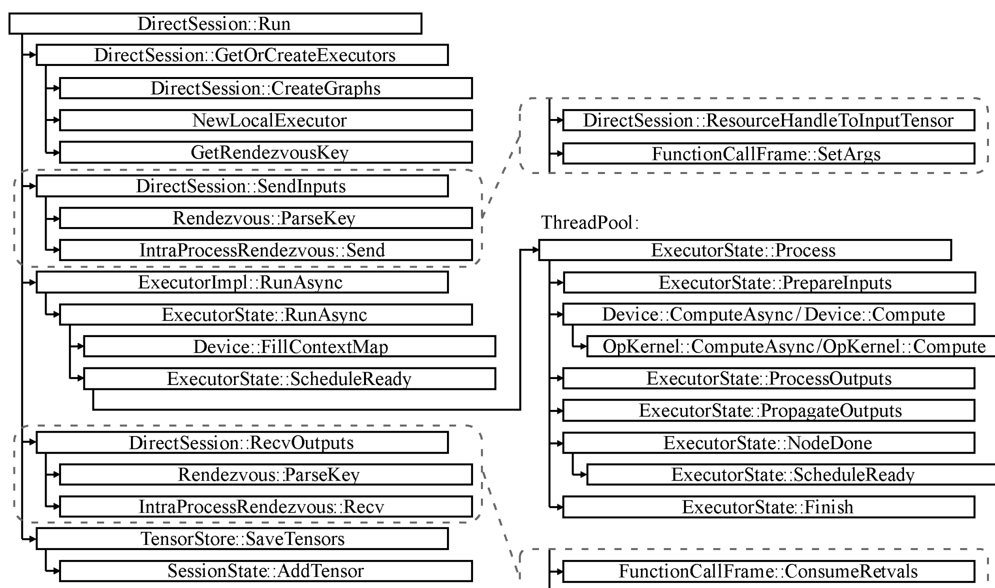


图 14-6　单机会话运行的主要流程

为了有效组织和调度会话运行过程，TensorFlow 提供了一组会话执行器和状态管理抽象。图 14-7 给出了单机会话运行过程相关抽象的 UML 类图。DirectSession 类已在之前章节中多次提及，它是单机运行模式下的应用层会话概念在 C++ 核心层的实现载体，其对象生命周期与会话一致。它的主要成员变量包括会话关联的设备指针、完整数据流图、执行器和状态管理对象，以及用于调度执行辅助对象方法的线程池指针等。DirectSession 类面向 C API 提供的主要公开方法包括会话的创建（Create）、扩展（Extend）、运行（Run）、部分运行（PRun）及关闭（Close），私有方法主要服务于图创建和会话运行过程。

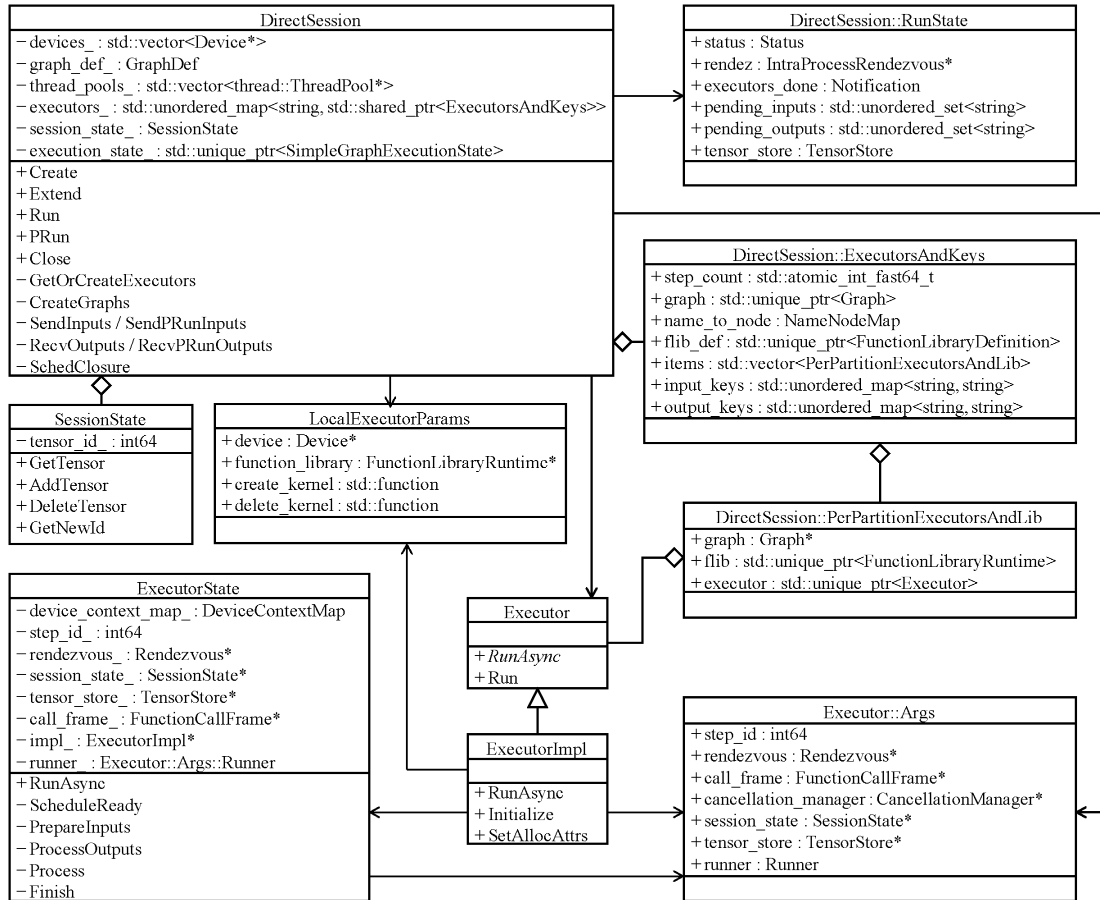


图 14-7　单机会话运行过程中相关抽象的 UML 类图（只列出关键成员变量与方法）

`DirectSession` 内嵌的 `RunState` 结构用于记录会话的一次特定运行相关的信息。其主要字段包括此次会话运行的状态、进程内通信所需的会合点对象指针、待运行的输入和输出节点名称集合，以及用于同步等待的 `Notification` 对象等。内嵌结构 `ExecutorsAndKeys` 用于记录会话运行关联的数据流图执行器对象集合、输入和输出张量对应的会合点键，以及节点名称到节点对象的映射等信息，以便管理数据流图与执行器、会合点等对象之间的关系。`ExecutorsAndKeys` 对象可以在同一会话的多次具有相同输入、输出的运行之间共享，从而节约计算和内存开销。内嵌结构 `PerPartitionExecutorsAndLib` 作为 `ExecutorsAndKeys` 结构中 `items` 向量的元素，用于记录切分后的每幅局部图与执行器对象的映射关系，以便在会话运行时调用合适的执行器执行局部图。`DirectSession` 的成员类型 `SessionState` 是一种张量管理类，用于以键值对方式存取会话涉及的 `Tensor` 对象集合。其辅助工具类 `TensorStore` 用于在 `SessionState` 对象中批量保存会话单次运行生成的输出张量。

Executor 类是会话执行器的抽象,它提供异步执行局部图的 RunAsync 虚方法及其同步封装版本 Run 方法。Executor 类的具体功能实现位于其子类 ExecutorImpl 中。内嵌结构 Args 用于为 RunAsync 方法提供每次独立运行的参数，其主要字段包括执行步 ID、会合点对象指针、用于管理函数调用上下文的 FunctionCallFrame 对象指针,以及用于张量持久化存储的 TensorStore 对象指针等。该结构中还定义了两个类型别名：用于表示待执行函数对象的 Closure 类型（std::function<void()>），以及用于为执行器实现函数调度功能、能够执行 Closure 对象的 Runner 类型（std::function<void(Closure)>）。Args 结构的 runner 字段为 Executor 对象赋予了在特定设备上调度执行函数的能力。为了方便创建与特定设备关联的 Executor 对象，TensorFlow 提供了 LocalExecutorParams 辅助结构。它记录了 Executor 对象所需的设备指针、自定义函数库，以及用于创建和删除核函数的函数对象等参数。

ExecutorState 类用于维护执行器的运行时状态，同时辅助 ExecutorImpl 类实现部分业务逻辑。ExecutorState 类以 Executor::Args 对象作为构造函数的参数，在构造时保存与 Executor:: Args 类似的成员变量。ExecutorState 对象在 ExecutorImpl::RunAsync 方法调用时构造，它提供的 RunAsync 公开方法被用做 ExecutorImpl::RunAsync 方法的内部实现。每个 ExecutorState 对象的生命周期关联到 ExecutorImpl::RunAsync 方法的一次调用。由此可以看出，ExecutorState 类与 DirectSession::RunState、DirectSession::ExecutorsAndKeys 结构的区别在于：后两者维护的是一次会话运行所提取的子图及其相关对象的状态，而 ExecutorState 维护的是子图切分之后、关联到特定设备的局部图的执行状态。

为了更加直观地说明上述概念的设计目标与应用场景，表 14-1 给出了单机会话运行所涉及的过程、抽象及数据流图类型的对应关系。

表 14-1　单机会话运行涉及的过程、抽象及数据流图类型对应关系

过　　程	数据流图类型	主体抽象	状态抽象
会话生命周期整体	全图	DirectSession	SessionState
会话单次运行	按计算语义提取的子图	ExecutorsAndKeys	RunState
会话在单个设备上的执行	按设备切分的局部图	Executor	ExecutorState

14.3.2　执行器获取

执行器获取是指创建或取得用于执行数据流图的 Executor 对象集合的过程，其入口方法是 DirectSession::GetOrCreateExecutors。该方法的输入参数给定了本次会话运行涉及的输入、输出张量名称和目标节点名称，以及用于调度执行辅助对象方法的线程池指针。主要输出参数是封装了 Executor 对象集合等成员的 ExecutorsAndKeys 对象指针。

为了节约执行器的创建开销，DirectSession 类提供了 ExecutorsAndKeys 对象缓存和复用机制。DirectSession 类的 executors_成员变量中保存了一组缓存的 ExecutorsAndKeys 对象集合。正如 GetOrCreateExecutors 方法名称所述，对于同一会话的多次具有相同输入、输出参

数的运行，只在第一次运行时创建 ExecutorsAndKeys 对象，此后直接返回缓存的对象指针。对于需要创建执行器的情况，GetOrCreateExecutors 方法首先调用 CreateGraphs 方法，对会话关联的数据流图进行待执行子图提取及面向设备的局部图切分（如 14.2 节所述）。然后，为每幅局部图准备一个带有其设备描述信息及核函数管理函数的 LocalExecutorParams 对象。基于这个对象，调用 NewLocalExecutor 函数，创建执行局部图所需的 Executor 对象，并将其加入 ExecutorsAndKeys 对象的 items 向量中。最后，遍历每个输入、输出张量名称，调用 GetRendezvousKey 函数，为它们设置进程内通信时所需的会合点键（如 13.2 节所述）。

这里简要介绍NewLocalExecutor函数的内部实现。该函数基于局部图对应的LocalExecutorParams 和 Graph 参数构造一个 ExecutorImpl 对象，然后调用新对象的 Initialize 方法对其进行初始化，最后以父类（Executor）指针的形式返回新对象。ExecutorImpl 的构造函数完成成员变量的初始化。ExecutorImpl::Initialize 方法用于准备执行局部图所需的数据结构，主要包括每个节点核函数对应的 OpKernel 对象，以及 ControlFlowInfo、FrameInfo 等辅助对象。其中，OpKernel 指针等数据流图对象的元信息被复制并封装到执行器的辅助类型——GraphView 和 NodeItem 中，作为 ExecutorImpl 对象的成员变量。GraphView 和 NodeItem 成员存储的信息看似冗余，但它们在执行器对象中以连续内存缓冲区和位元组等高效的方式存储，能够在运行时快速地访问，从而减少随机访存和间接访存的开销。

14.3.3 输入数据填充

在获取执行器之后，DirectSession::Run 方法将为待运行的数据流图填充输入数据（即张量）。通过对子图提取过程的分析，我们已经知道输入数据填充有两种实现方式：基于 SendOp 和 RecvOp，使用会合点机制传输数据；基于 ArgOp 和 RetvalOp，借助 FunctionCallFrame 对象传输数据。后者相比前者，适用范围有限，但是它能避免一部分字符串解析和内存访问开销，因此具有更好的性能。

对于单机会话，TensorFlow 1.1 及更早版本默认使用会合点方式，而 1.2 版本之后则改用 FunctionCallFrame 方式。虽然会合点方式已经在 TensorFlow 1.2 的单机会话 Run 方法中弃用，但是它仍然应用于 PRun 方法及分布式会话，因此有必要了解。这里分别对这两种方式进行介绍。

- 会合点方式：DirectSession::Run 方法首先创建用于记录本次会话运行状态的 RunState 对象和进程内通信所需的会合点对象，随后调用 DirectSession::SendInputs 方法，为待执行的数据流图填充输入张量（在 1.2 版本以后，该方法更名为 SendPRunInputs）。DirectSession::SendInputs 方法的输入参数包括来自 DirectSession::Run 方法参数的 inputs 向量，以及本次会话运行所需的 ExecutorsAndKeys 和 IntraProcessRendezvous 对象指针。该方法的逻辑相对简单：遍历 inputs 中的所有输入张量，根据其名称在 ExecutorsAndKeys 对象中找到对应的会合点键。对于每个会合点键，调用 Rendezvous::ParseKey 方法，将其转换为 Rendezvous::ParsedKey 结构保存。然后以 ParsedKey 对象

作为数据标识、以输入张量作为待发送数据，调用 IntraProcessRendezvous::Send 方法发送张量。之前子图提取时所插入的 RecvOp 将在图运行过程中调用会合点的 RecvAsync 方法，实现张量接收。

- FunctionCallFrame 方式：DirectSession::Run 方法首先创建用于记录本次会话运行状态的 RunState 对象，以及代表函数调用上下文的 FunctionCallFrame 对象；然后遍历来自 DirectSession::Run 方法参数的 inputs 向量，通过 DirectSession::ResourceHandleToInputTensor 方法找到每个资源句柄对应的输入张量；接下来调用 FunctionCallFrame::SetArgs 方法，将输入张量集合作为函数上下文参数赋予 FunctionCallFrame 对象。之前子图提取时所插入的 ArgOp 将在图运行过程中调用 FunctionCallFrame::GetArg 方法，读入函数上下文中的张量，实现输入数据填充。

14.3.4 图运行

数据流图运行是会话运行的核心步骤，它是指一组执行器在各个设备上依照数据流图定义的依赖关系加载运行图上的操作节点核函数，对输入数据实施计算，生成输出数据的过程。这一过程允许多线程、多设备的并发执行，以便提升计算效率。图运行的入口方法是 ExecutorImpl::RunAsync。在图运行之前，需要进行必要的准备工作。

图运行前的第一项准备工作是创建 ExecutorBarrier 对象，该对象用于实现不同局部图执行器对象的障栅同步操作（barrier），从而保证多线程并发的图运行逻辑在所有线程均完成后才进入后续流程。ExecutorBarrier 类的实现原理如下。

(1) 父线程构造 ExecutorBarrier 对象时，指定参与障栅同步的子线程数量，该数量会保存到一个计数器变量。

(2) 父线程调用 Notification::WaitForNotification 方法阻塞等待通知事件。

(3) 各个子线程到达同步点后，互斥地对共享对象中的计数器变量执行自减操作。

(4) 当某个线程发现共享计数器值自减后变为 0，则认定自己是最后一个到达同步点的线程，此时需要调用 Notification::Notify 方法，通知父线程结束等待，继续执行。

DirectSession::Run 方法集成障栅同步操作的做法如下。

(1) 在图运行逻辑中，触发 ExecutorBarrier 计数器自减的 ExecutorBarrier::WhenDone 操作以回调函数的方式传入 ExecutorImpl::RunAsync 方法，供子线程在工作完成之后调用。

(2) 用于同步等待的 Notification::WaitForNotification 方法被封装入 DirectSession::WaitForNotification 方法，由父线程在图运行之后调用。

图运行前的另一项准备工作是初始化 Executor::Args 对象。对于会话的同一次运行，Args 对象在所有 Executor 对象间共享，以便各个执行器共享会合点、张量缓存等资源。Runner 类型的 runner 字段作为执行函数调度功能的函数对象，使用封装了 DirectSession::SchedClosure

方法的匿名函数进行初始化。SchedClosure 方法将传入的函数对象调度到 DirectSession 管理的线程池内执行。

完成必要的准备之后，DirectSession::Run 方法将遍历 ExecutorsAndKeys 对象保存的所有 Executor 对象，为每幅局部图分别调用图运行方法 ExecutorImpl::RunAsync。该方法为当前 Executor 创建一个 ExecutorState 对象，用于维护本次运行状态。继而调用 ExecutorState::RunAsync 方法，启动图运行。ExecutorState::RunAsync 方法首先调用局部图对应设备的 Device::FillContextMap 方法，初始化 ExecutorState 内部的设备上下文（DeviceContext）对象集合。对于 CPU 设备，初始化的实现为空；对于 GPU 设备，初始化工作用于建立图上节点到 GPU 流 ID 的映射。接下来，ExecutorState::RunAsync 方法将图上就绪的操作节点加入 ready 向量，并调用 ExecutorState::ScheduleReady 方法对这些操作实施调度。ScheduleReady 方法遍历就绪节点向量，依据每个节点对应的操作核函数的 kernel_is_expensive 属性执行不同的逻辑：对于标记为开销较大的节点，使用从 Executor::Args 对象传入的 Runner 函数调用 ExecutorState::Process 方法，使得操作被加载到线程池的空闲线程上执行。对于标记为开销较小的节点，将其加入作为参数传入的 inline_ready 队列——这些节点将在 ExecutorState::Process 方法中以内联方式在同一线程中执行，以节约线程分配开销。

数据流图节点操作在对应设备上的执行是由 ExecutorState::Process 方法发起的。尽管其输入参数只包含一个待执行节点，然而由于内联执行机制的存在，该方法还会执行相关的内联节点。Process 方法内置一个 inline_ready 节点队列，该方法的参数给定的待执行节点作为首个元素加入该队列。Process 方法运行时，遍历队列中的节点，依次执行。一个节点执行完毕后，级联调用 ExecutorState::NodeDone 和 ExecutorState::ScheduleReady 方法，在队列中添加新的内联节点。直到 inline_ready 队列中的节点全部执行完毕，方法才会退出。退出之前，Process 方法会调用 ExecutorState::Finish 方法，通过该方法调用作为 ExecutorState::RunAsync 参数传入的回调函数，从而触发障栅同步。

具体到特定操作节点的执行，ExecutorState::Process 方法首先调用 ExecutorState::PrepareInputs 方法，为节点准备输入张量。然后依据节点操作核函数的 kernel_is_async 属性，调用对应设备的 Device::Compute 或 Device::ComputeAsync 方法，同步或异步地执行操作核函数提供的 OpKernel::Compute 或 OpKernel::ComputeAsync 方法，实现节点的计算逻辑。接下来调用 ExecutorState::ProcessOutputs 方法，对输出张量加以收集和处理。最后调用 ExecutorState::PropagateOutputs 方法，将输出张量传递到本节点对应的目标节点。对于异步执行操作核函数的情况，ProcessOutputs 和 PropagateOutputs 方法是在 Device::ComputeAsync 方法的完成回调函数中调用的。

至此，定义在应用程序代码中的、高度抽象数据流图经过一系列分解和重组，落实到了具体的计算设备。

14.3.5　输出数据获取

所有执行器到达障栅同步点之后，回调函数将对 RunState 对象保存的会话运行状态进行更新，随后进行数据流图的输出数据获取。输出数据获取同样有会合点和 FunctionCallFrame 两种实现方式，下面分别介绍。

- **会合点方式**：本方式的入口方法是 DirectSession::RecvOutputs（在 1.2 版本以后，该方法更名为 RecvPRunOutputs）。其输入参数包括来自 DirectSession::Run 方法参数的输出节点名称向量，以及会话运行所使用的 ExecutorsAndKeys 等对象指针。输出参数是会话运行生成的张量集合，它将为 DirectSession::Run 方法的 outputs 参数提供输出。DirectSession::RecvOutputs 的逻辑同 DirectSession::SendInputs 类似。它需要在 ExecutorsAndKeys 对象中找到输出张量对应的会合点键，并使用 Rendezvous::ParseKey 方法对键进行解析。然后以解析后的 ParsedKey 对象作为数据标识，通过 IntraProcessRendezvous::Recv 方法接收张量。之所以能够在这里接收到输出张量，是因为之前子图提取时所插入的 SendOp 已在图运行过程中调用了会合点的 Send 方法，完成了张量发送。
- **FunctionCallFrame 方式**：本方式由 FunctionCallFrame::ConsumeRetvals 方法实现。该方法遍历函数上下文中的返回值（即输出张量），将其逐个赋予临时向量。DirectSession::Run 方法随后依据待输出张量的索引顺序，将临时向量中的张量移动到 outputs 参数指向的输出向量，完成输出数据获取。之所以能够在函数上下文的返回值中取得输出张量，是因为之前子图提取时所插入的 RetvalOp 已在图运行过程中调用了 FunctionCallFrame::SetRetval 方法，在返回值中写入了输出张量。

14.3.6　张量保存

TensorFlow 允许在会话中对张量进行缓存和复用，这是借助 DirectSession 类的 SessionState 成员和 RunState 结构的 TensorStore 成员完成的。TensorStore 对象的 AddTensor 方法用于添加待缓存的张量名称，该方法通常在 GetSessionHandleOp 操作中调用。

单机会话运行的最后一个步骤是调用 TensorStore::SaveTensors 方法，将本次运行输出的张量进行选择性地保存。TensorStore::SaveTensors 方法遍历本次运行的所有输出节点名称，如果其存在于 TensorStore 对象已记录的名称集合，则调用 SessionState::AddTensor 方法保存对应的张量。同一会话再次运行时，输入数据填充过程可以级联调用 DirectSession::ResourceHandleToInputTensor 和 SessionState::GetTensor 方法，取得此前缓存的张量。

14.4　分布式会话运行

在 TensorFlow 的分布式运行模式下，会话的运行不仅可以是跨单机多设备的，也可以是跨多机的。对于应用开发者而言，分布式应用的运行除需使用专门的 Supervisor、Optimizer 等抽象

外，会话本身的接口同单机运行模式保持兼容。因此，在 TensorFlow C++ 核心中，分布式会话机制亦同单机会话保持一致的接口语义。为了有效管理分布式会话运行，TensorFlow 设计了一种主–从（master-worker）模型，将会话的运行时抽象和流程分解到不同角色的运行时实体上。多个会话实例及同一会话的各个组件在 gRPC 通信机制的辅助下，能够实现跨进程协同。本节将介绍分布式会话运行的原理与设计。

14.4.1 主–从模型

主–从模型包含 master、worker 和 client 角色及其对应的功能组件。在核心层代码中，master 与 worker 是管理会话与数据流图的实体。master 组件负责管理分布式会话的生命周期及数据流图执行任务的分配，worker 组件负责管理数据流图的局部图在特定进程、特定设备上的执行。master 与 worker 角色并没有绑定到两类不同的进程，它们之间也不存在固定的映射关系。每个 TensorFlow 任务（一般对应到一个进程）均具备同时履行 master 与 worker 两种角色的能力，这体现在每个进程运行时均通过 gRPC 提供 master 和 worker 两类服务，并对集群中的其他进程开放访问。与 master 和 worker 角色相对应，TensorFlow 把定义数据流图、创建并运行会话的运行时实体定义为 client 角色。client 程序一般是指使用 TensorFlow API 开发的应用程序，但 client 角色的功能实现代码仍位于 TensorFlow 核心层。对于当前的 TensorFlow 开源实现，client 角色往往与 master、worker 运行在同一进程内。

需要注意的是，TensorFlow 核心层的主–从模型同应用层 API 提供的 PS-worker 编程模型并非同一概念。两个模型中同名的 worker 概念具有不同的含义，不能认为 PS 或 chief worker 任务属于 master 角色、worker 任务属于 worker 角色。master 和 worker 角色作为逻辑概念，对 TensorFlow 任务实体而言并非确定或唯一，而是与具体的会话运行过程相绑定的，并允许动态变化。尽管 Google 在 TensorFlow 架构文档（详见 https://www.tensorflow.org/extend/architecture）中给出了一种“1 client – 1 master – 多 worker”的典型图示，然而我们需要知道，这并非 TensorFlow 分布式应用的唯一布局模式。事实上，上层抽象对核心层机制的使用方式决定了应用层语义到主–从模型的映射关系。Python API 等上层接口提供的抽象为分布式会话带来灵活的配置能力，允许应用开发者编写不同并行模式的程序；二次开发者亦可以在 C API 基础上设计更加多样的编程模型，充分发挥主–从模型的可扩展能力。

这里以一个典型的 PS-worker 模式应用程序——TensorFlow 自带的 mnist_replica.py 为例，说明主–从模型中的角色在应用任务中的体现。图 14-8 给出了该应用的 1 PS + 2 worker 运行实例，其中主–从模型的角色名称以方括号标识，PS-worker 模型的任务名称以花括号标识。所有任务的进程均由同一份程序（包括 Python 脚本、解释器及 TensorFlow 运行时库）启动，因此它们均具备同时履行 master、worker 和 client 三种角色的能力。然而，PS 任务在启动之后随即进入阻塞等待状态，其 client 代码不再驱动会话的运行，master 服务也不被其他任务调用，因此可认为 PS 任务只作为 worker 角色存在。对于 worker 任务，无论它是否为 chief worker，均具有独立的会话对象，均由其 client 代码调用 master 服务，驱动会话的运行（`MasterService::RunStep` 接口）。

会话运行时，依据数据流图切分结果，master 组件有可能调用本任务或 PS 任务的 worker 服务，运行不同的局部图（`WorkerService::RunGraph` 接口）。PS 和 worker 任务上的图运行时，亦会通过 worker 组件发起进程间通信，实现参数交换（`WorkerService::RecvTensor` 接口）。因此，两个 worker 任务各自具有三种逻辑角色。但以任务自身视角观察，符合“1 client – 1 master – 多 worker”的视图。

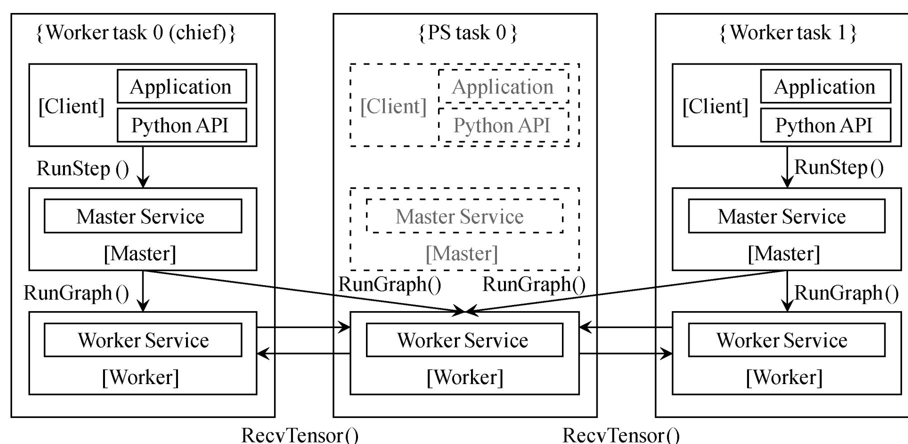


图 14-8 主–从模型中的角色在典型 PS-worker 应用任务中的体现

14.4.2 主要抽象

图 14-9 给出了分布式会话涉及的主要抽象的 UML 类图。作为分布式运行时驱动框架的 gRPC 服务，我们已在 13.3 节中介绍过。在该框架中，master 和 worker 服务的接口代码由 master_service.proto 和 worker_service.proto 文件定义，它们经过 Protocol Buffers 编译生成相应的 C++ 代码。TensorFlow 源代码包中的 `MasterService` 和 `WorkerService` 类对部分自动生成的方法进行了覆盖或特化实现。`MasterInterface` 和 `WorkerInterface` 类分别定义了 master 和 worker 组件提供的功能接口，这些 C++ 接口与相应角色的 gRPC 服务接口基本一一对应。`GrpcRemoteMaster` 和 `Grpc RemoteWorker` 类继承了这些 C++ 接口，作为两种服务在调用方的存根类使用，其主要工作是向同名的远程 gRPC 函数发起请求。`GrpcMasterService` 和 `GrpcWorkerService` 类则是两种服务的顶层实现类，它们调用 master 与 worker 组件，实现 gRPC 函数的功能，并输出响应消息。

master 与 worker 角色的功能实现类分别是 `Master` 类和 `Worker` 类。一方面，两个类的方法可以被同名的 gRPC 实现函数直接或间接调用，以便响应远程请求。另一方面，两种角色均具有本地调用路径的优化设计，即在访问同一进程内的实现对象时，无须经由 gRPC 机制访问回环网络设备，而是直接调用本地对象的方法。具体而言，对于响应远程请求的情况：(1) `Master` 对象的方法可以直接被 `GrpcMasterService` 类的成员方法调用；(2) `Worker` 类具有名为 `GrpcWorker` 的子类，专门提供给 `GrpcWorkerService` 类使用。它以更优化的方式覆盖实现了 `RecvTensorAsync`

方法，可以避免 gRPC 通信时数据序列化的开销。对于响应本地请求的情况：(1) master 组件的本地调用路径优化由 `LocalMaster` 类实现，该类继承了 `MasterInterface` 类定义的接口，并将请求转发给本地的 `Master` 对象执行，从而绕过了 `GrpcRemoteMaster` 类。`LocalMaster` 类作为中介类型存在的意义，是将 `Master` 类的异步方法经由 `Notification` 机制封装为同步方法，从而简化本地方法调用的逻辑。(2) `Worker` 类继承了 `WorkerInterface` 类定义的接口。当 master 组件访问本地 `Worker` 对象时，可以直接调用 `Worker` 对象的方法，无须借助 `GrpcRemoteWorker` 类。

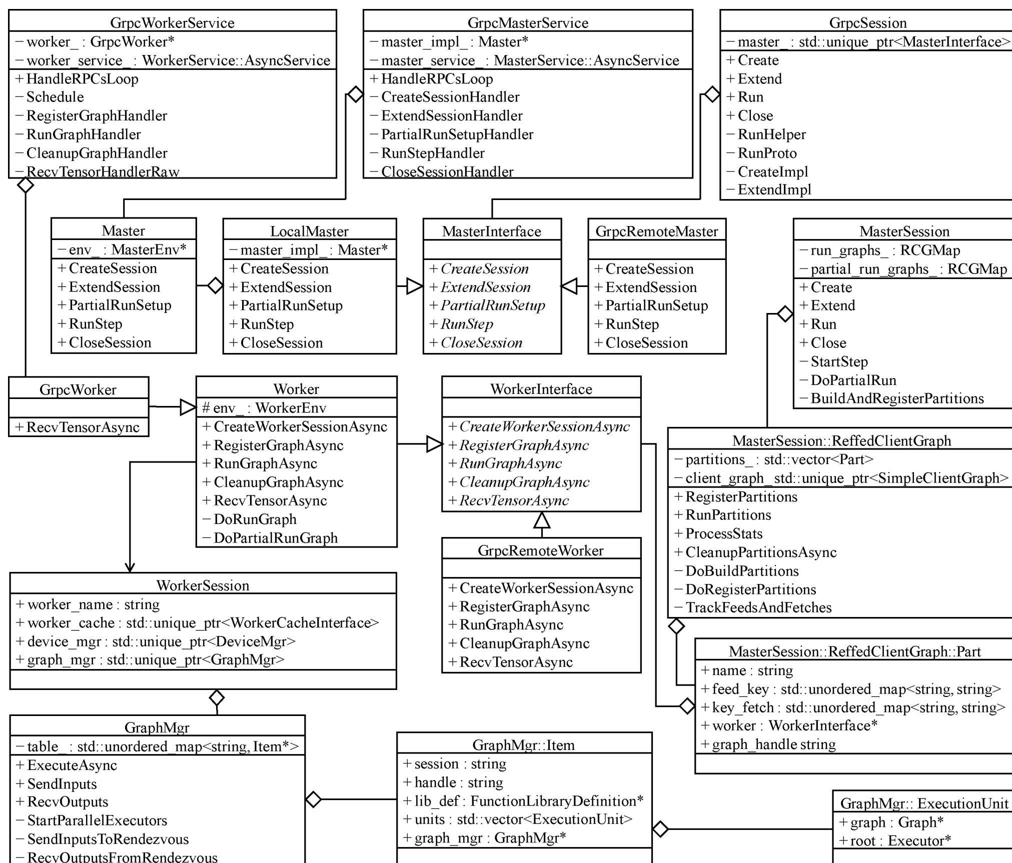


图 14-9　分布式会话运行过程中相关抽象的 UML 类图（只列出关键成员变量与方法）

分布式会话抽象在 TensorFlow 核心层被分解为 `GrpcSession`、`MasterSession` 和 `WorkerSession` 等抽象，它们各自的功能介绍如下。

- ❑ `GrpcSession` 类是会话的上层（调用层）抽象，其实例运行于 client 组件中。它同单机会话的 `DirectSession` 类一样，继承了 `Session` 父类，代表完整数据流图上的整个会话生命周期。`GrpcSession` 类提供会话的创建、扩展、运行等方法，这些方法的实现由私有的 `MasterInterface` 指针指向的对象完成。当待调用的 master 组件位于会话所处的同一

进程时，指针指向 LocalMaster 对象；否则，指针指向 GrpcRemoteMaster 对象。

- MasterSession 类是会话的下层（执行层）抽象，其实例运行于 master 组件中。作为会话操作的实施者，该类封装了会话运行所需的数据流图、设备等资源指针，以及 MasterSession::RunState 等状态抽象。内嵌类 MasterSession::ReffedClientGraph 是针对 SimpleClientGraph 对象指针的一种带有引用计数的封装类型，主要用于 master 组件驱动数据流图的运行。内嵌结构 MasterSession::ReffedClientGraph::Part 用于管理切分后的数据流图。
- WorkerSession 是一种在 worker 组件中使用的简单数据结构，本身并不提供成员方法。它封装了会话在本 worker 组件中执行时所需调用的几种外部对象（如设备管理器、数据流图管理器）的指针，以便于 Worker 对象在会话运行时调用外部对象的方法。

GraphMgr 类用于在 worker 组件中管理数据流图，它提供数据流图注册、执行、注销以及输入和输出张量访问等方法。内嵌结构 GraphMgr::Item 记录了每幅图的元信息，其指针集合作为 GraphMgr 类的成员（table_）保存。另一内嵌结构 GraphMgr::ExecutionUnit 记录了切分后的局部图所关联的设备、执行器等对象指针，其实例集合作为 GraphMgr::Item 结构的成员（units）保存。此外，TensorFlow 还设计了 SessionMgr 类，用于在 worker 组件中管理 WorkerSession 对象。它存储着 worker 组件所使用的 WorkerSession 对象指针的集合，提供基于数据流图句柄或执行步 ID 获取 WorkerSession 指针的方法。

14.4.3　client创建会话

在 C API 层面，单机会话与分布式会话的创建具有相同的入口，即 TF_NewSession 函数。该函数判断应当创建单机会话（DirectSession）还是分布式会话（GrpcSession）的依据是 SessionOptions 结构的 target 字段。这个结构是由 Python API 等上层接口封装到 TF_SessionOptions 参数中的，结构内的 target 字段用于指定会话待连接的 TensorFlow 运行时位置：为空时表示使用本地运行时，为形如“grpc://[主机名]:[端口]”的字符串时表示使用分布式运行时。TF_NewSession 函数通过全局的 NewSession 函数调用会话工厂基类 SessionFactory 的 GetFactory 静态方法，该方法依据 target 字段选择性地返回对应的会话工厂类。全局 NewSession 函数随后调用工厂类的 NewSession 方法，进而级联调用相应会话类的 Create 方法，实现会话创建。

图 14-10 给出了 client 组件创建会话的主要流程。对于分布式会话，GrpcSession::Create 静态方法首先使用 SessionOptions 参数构造一个 GrpcSession 对象。然后，调用 LocalMaster::Lookup 静态方法，检查 target 字段指定的 master 组件是否位于当前进程中。如果是，则使用 LocalMaster 对象封装的本地 Master 对象；否则依次调用 NewHostPortGrpcChannel 和 NewGrpcMaster 全局函数，创建用于访问远程 Master 对象的 GrpcRemoteMaster 存根对象。最后，调用 GrpcSession::SetRemoteMaster 方法，将本地或存根 master 组件的指针保存在新建的 GrpcSession 对象中，以供后续使用。

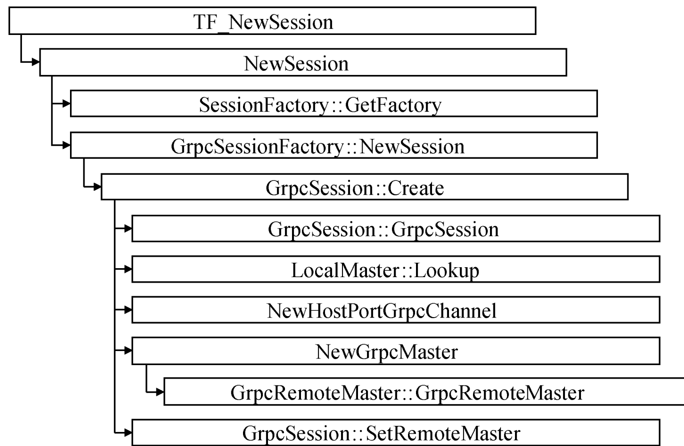


图 14-10　client 创建会话的主要流程

14.4.4　client请求图运行

分布式会话的运行涉及 client – master – worker 三个调用层次，其中 client 组件是会话运行的起点。应用层代码通过多语言 API 间接调用 `GrpcSession::Run` 方法，启动会话的运行。`GrpcSession::Run` 作为入口方法，具有同单机会话运行类似的签名：`Status GrpcSession::Run (const RunOptions& run_options, const std::vector<std::pair<string, Tensor>>& inputs, const std::vector<string>& output_tensor_names, const std::vector<string>& target_node_names, std::vector<Tensor>* outputs, RunMetadata* run_metadata)`。其中，主要输入参数 `inputs`、`output_tensor_names`、`target_node_names` 分别是本次运行涉及的输入张量集合以及输出和目标节点名称向量，主要输出参数 `outputs` 是本次运行生成的张量集合。与单机会话不同的是，`GrpcSession::Run` 的调用栈并不实现会话运行的全流程，而是负责向 master 组件发出数据流图运行请求。

图 14-11 给出了 client 组件请求图运行的主要流程。`GrpcSession::Run` 方法将参数转发给通用性更强的 `GrpcSession::RunHelper` 方法。`GrpcSession::RunHelper` 方法首先调用 `MasterInterface` 类或其本地优化子类的 `CreateRunStepRequest` 和 `CreateRunStepResponse` 方法，创建访问 master 服务 `RunStep` 接口所需的请求与响应消息包装类（`MutableRunStep RequestWrapper`、`MutableRunStepResponseWrapper`）的子类对象，并使用接口参数为请求对象的成员赋值。TensorFlow 针对三种组件相对位置的不同组合情况，提供了 gRPC 消息包装类的多种子类，其目的是优化请求与响应中的张量存储方式，尽可能避免数据复制开销。这些包装类的实现参见 tensorflow/core/distributed_runtime/message_wrappers.h 文件。



图 14-11　client 请求图运行的主要流程

接下来，`GrpcSession::RunHelper` 方法将消息包装对象传递给 `GrpcSession::RunProto` 方法。后者会调用 `GrpcSession` 私有的 `master_` 指针的 `RunStep` 方法，正式发起图运行请求。对于远程运行和本地优化的情况，`master_` 指针分别指向 `GrpcRemoteMaster` 和 `LocalMaster` 对象。远程运行数据流图时，`GrpcRemoteMaster::RunStep` 方法会调用 gRPC 自动生成的存根类 `grpc::MasterService::Stub` 的 `RunStep` 方法，向 master 组件所在的进程发起图运行请求。进入本地优化路径时，`LocalMaster::RunStep` 方法会调用当前进程 `Master` 对象的 `RunStep` 方法，其后续调用栈等同于下一节介绍的 master 驱动图运行流程。

最后，`GrpcSession::RunHelper` 方法同步等待 `GrpcSession::RunProto` 方法的返回。返回之后，它将提取响应消息中的输出张量，通过指针交换方式赋值给上级调用者传入的输出参数，完成数据流图的一次分布式执行步骤。

14.4.5　master驱动图运行

master 组件的 gRPC 服务接收到来自 client 的 `RunStep` 请求后，将会驱动数据流图的运行。图 14-12 给出了这一过程的主要流程。本过程以 `GrpcMasterService::RunStepHandler` 方法为起点，该方法从 gRPC 调用中提取出请求消息，交由服务关联的 `Master` 对象的 `RunStep` 方法执行。后者具有一个匿名的回调函数参数，用于在方法执行完毕后向 client 组件发送 gRPC 响应（即 `Call::SendResponse` 方法）。`Master::RunStep` 方法基于请求消息中的会话句柄字段，为本次图运行找到对应的 `MasterSession` 对象。然后将 `MasterSession::Run` 方法连同传入的回调函数一并封装到匿名函数对象中，交由 `SchedClosure` 函数调度至全局线程池中执行。

`MasterSession::Run` 方法依据 `RunStep` 请求中的 `partial_run_handle` 字段值，将请求转发给本对象的 `DoPartialRun` 或 `DoRunWithLocalExecution` 方法执行。对于该参数为空的情况，`DoRunWithLocalExecution` 方法是图运行过程的核心驱动者。该方法内部可分为子图提取、图切分与注册、图运行，以及状态处理和资源清理等步骤。子图提取由 `MasterSession::StartStep` 方法完成，该步骤类似于单机会话数据流图创建过程中的子图提取。它以包含待运行子图的输入、输出节点信息的 `BuildGraphOptions` 结构为输入，通过调用 `SimpleGraphExecutionState::`

BuildGraph 方法创建子图的 SimpleClientGraph 对象，然后将子图封装到 MasterSession::ReffedClientGraph 对象并返回。

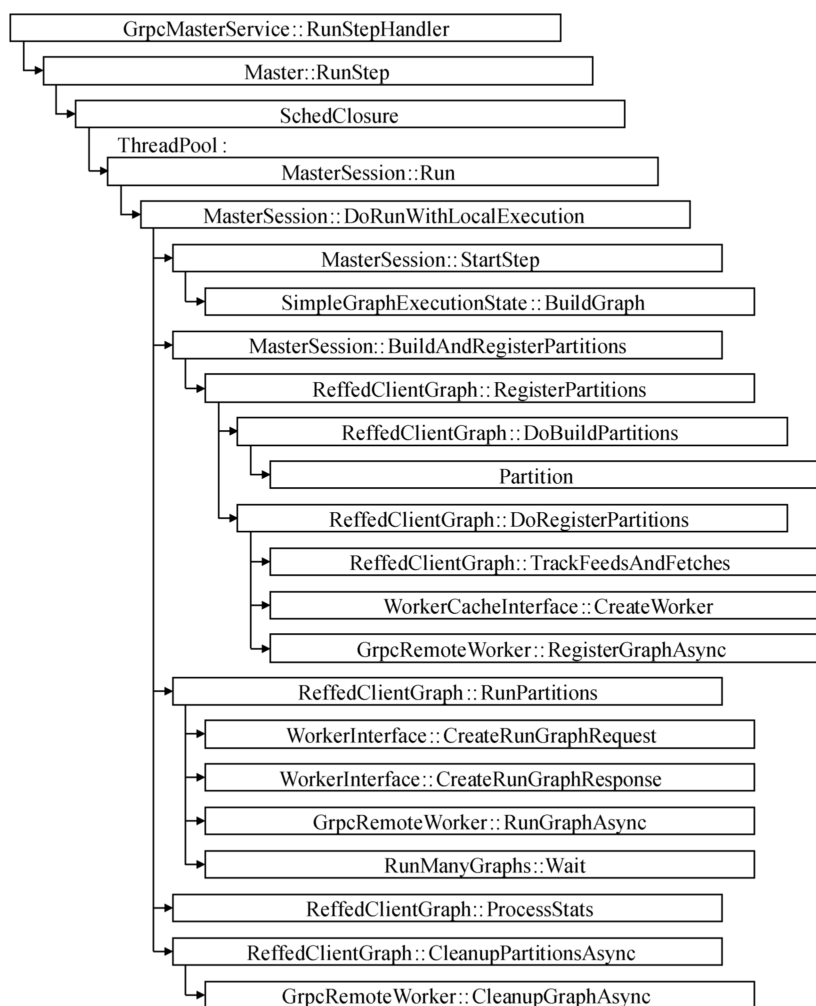


图 14-12　master 驱动图运行的主要流程

图切分与注册由 MasterSession::BuildAndRegisterPartitions 方法完成。该方法构建切分操作所需的 PartitionOptions 参数之后，调用 ReffedClientGraph 对象的 RegisterPartitions 方法，先后执行图切分（DoBuildPartitions）和图注册（DoRegisterPartitions）这两个步骤。

- 图切分步骤与单机会话的图切分类似，由 Partition 全局函数完成。不同的是，此时每幅局部图所映射的设备有可能是注册到不同的任务进程、位于不同的服务器上的。

- 图注册的目的是将所有局部图的运行任务分配到各自对应的 worker 组件。该步骤遍历每幅局部图，首先调用 `ReffedClientGraph::TrackFeedsAndFetches` 方法，将图的输入、输出节点信息记录到 `MasterSession::ReffedClientGraph::Part` 结构中。然后调用 `WorkerCacheInterface::CreateWorker` 方法，为图的运行准备 worker 组件。`CreateWorker` 方法返回的指针有可能指向事先缓存的、为本地调用路径优化过的 `GrpcWorker` 对象，也有可能指向作为访问远程 worker 组件的存根而新建的 `GrpcRemoteWorker` 对象。以后者为例，图注册步骤会调用存根对象的 `RegisterGraphAsync` 方法，向 worker 服务发出 gRPC 请求，完成待执行局部图在 worker 组件中的注册。

完成图切分与注册之后，`DoRunWithLocalExecution` 方法将调用 `MasterSession::ReffedClientGraph::RunPartitions` 方法，驱动数据流图运行。`RunPartitions` 方法首先调用 `WorkerInterface` 类的 `CreateRunGraphRequest` 和 `CreateRunGraphResponse` 方法，为 `RunGraph` 接口的 gRPC 调用创建请求与响应消息包装对象。这些对象同 client 组件访问 `RunStep` 接口时创建的消息包装对象类似，主要用于优化数据存储，避免不必要的复制开销。随后，`RunPartitions` 方法遍历记录在 `MasterSession::ReffedClientGraph::Part` 对象中的每个 worker 组件指针，调用其 `RunGraphAsync` 方法。对于远程运行局部图的情况，这时调用的将是 `GrpcRemoteWorker::Run GraphAsync` 方法。该方法对 worker 服务发起 `RunGraph` 的 gRPC 请求，从而驱动已注册的局部图在 worker 组件上的异步运行。最后，`RunPartitions` 方法会调用辅助类型 `RunManyGraphs` 的 `Wait` 方法，同步等待所有 `RunGraph` 调用的完成，并获取输出张量。`RunManyGraphs` 类的功能及实现类似于单机会话中的 `ExecutorBarrier` 类。

图运行完成后，`DoRunWithLocalExecution` 方法将调用 `MasterSession::ReffedClientGraph::ProcessStats` 和 `MasterSession::ReffedClientGraph::CleanupPartitionsAsync` 方法，进行状态处理和资源清理。`ProcessStats` 方法将数据流图操作节点等对象的状态标记为已完成，并记录当前执行步的执行时间。`CleanupPartitionsAsync` 方法则会调用本地或远程 worker 组件的 `CleanupGraphAsync` 方法，通知其清理当前执行步涉及的相关资源，以节约内存开销。

14.4.6 worker实施图运行

worker 组件的 gRPC 服务接收到来自 master 的 `RunGraph` 请求后，将会实施数据流图的运行。图 14-13 给出了这一过程的主要流程。本过程的起点是 `GrpcWorkerService::RunGraphHandler` 方法。作为 worker 服务的接口之一，`RunGraph` 接口的实现机制同 13.3.3 节介绍的 `RecvTensor` 接口有类似之处，即通过逐级封装的回调函数实现 gRPC 服务器端逻辑的异步处理（图中以虚线箭头表示匿名回调函数的定义位置）。同时，分布式会话的局部图执行逻辑复用了单机会话的部分代码，使得 TensorFlow 的数据流图底层计算机制具有统一的实现。

`GrpcWorkerService::RunGraphHandler` 被 gRPC 事件机制触发后，旋即调用 `GrpcWorkerService::Schedule` 方法，将封装了本地 `GrpcWorker` 对象 `Worker::RunGraphAsync` 方法的匿名函数调度到本进程的线程池中排队执行。`RunGraphAsync` 的输入参数包含一个封装了 `Call::`

SendResponse 方法的完成回调函数，它将在 RunGraphAsync 执行完毕之后调用，以便向 master 组件发送 gRPC 响应。

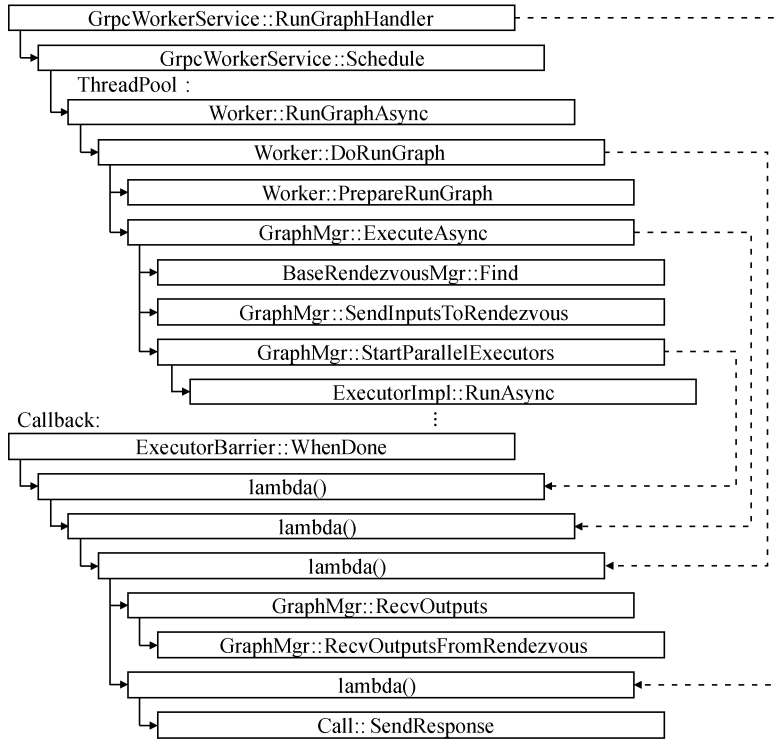


图 14-13　worker 实施图运行的主要流程

Worker::RunGraphAsync 方法依据请求消息中的 is_partial 字段值，将请求转发给本对象的 DoPartialRunGraph 或 DoRunGraph 方法执行。对于执行整个局部图的常规情况，图运行逻辑由 DoRunGraph 方法执行。该方法首先调用 Worker::PrepareRunGraph 方法，从 gRPC 请求中提取图运行所需的参数。然后调用本地 SessionMgr 管理的 GraphMgr 对象的 ExecuteAsync 方法，异步执行已注册的局部图。ExecuteAsync 方法的输入参数亦包含一个完成回调函数，该回调函数用于在图运行完毕后收集输出张量，写入 gRPC 响应消息。

GraphMgr::ExecuteAsync 方法执行局部图的流程与单机会话的图运行过程类似，其中输入数据填充和输出数据获取使用的是会合点方式。首先，通过 BaseRendezvousMgr::Find 方法得到当前执行步对应的会合点对象。然后，调用 GraphMgr::SendInputsToRendezvous 方法向会合点发送输入数据。接下来，调用 GraphMgr::StartParallelExecutors 方法，启动局部图在特定计算设备上的执行（可参考 14.3.4 节对 ExecutorImpl::RunAsync 等方法的详细说明）。最后，在 ExecutorBarrier 对象完成障栅同步操作之后进入完成回调函数，调用 GraphMgr::RecvOutputs 方法，从会合点接收输出数据。至此，worker 组件实现了局部图的一个执行步在分布式会话中的运行。

14.5 操作节点执行

无论在何种运行模式下，数据流图操作节点的执行最终需要落实到具体的计算设备。本书已在 12.2.3 节介绍了操作节点相关的主要数据结构，说明了基于方法链的操作定义与注册机制；并在 12.4.1 节介绍了操作节点支持异构加速器的原理，说明了基于 C++ 模板的异构设备核函数实现技巧。在此基础上，我们可以进一步分析会话运行过程中操作节点及其核函数的执行逻辑。

TensorFlow 提供的操作种类众多，既包括面向矩阵和张量计算的各种代数算法操作，也包括用于数据填充、通信和排队的多种流程控制操作。这里以神经网络算法中常用的矩阵乘法操作（`MatMulOp`）为例，介绍其执行原理。

14.5.1 核函数抽象

在 TensorFlow C++ 核心层，操作节点执行过程的本质是节点对应的核函数执行过程，因此有必要了解核函数抽象及其在会话中的地位。本节不再重复陈述前面介绍过的核函数关键抽象，而是结合数据流图的运行过程，说明核函数对象的创建过程及运行时辅助数据结构的设计。

从计算语义的角度看，数据流图的操作节点是设备无关的。但数据流图执行时，操作核函数必须在特定的设备上运行。因此，核函数对象的创建只有在面向设备的局部图切分和节点放置过程完成之后才能执行。在会话执行器获取过程（例如单机会话的 `DirectSession::GetOrCreateExecutors` 方法）中，辅助结构 `LocalExecutorParams` 的 `create_kernel` 函数指针会被赋予一个匿名函数，用于实现核函数对象的创建。该匿名函数关联到了执行器对应的计算设备。会话运行时，`ExecutorImpl::Initialize` 方法会对数据流图上的每个操作节点分别调用 `create_kernel` 函数。这时创建的核函数对象将是对应操作在特定设备上的特化版本，即基于 C++ 模板特化机制，为特定设备类型独立编写的版本。

`create_kernel` 函数之所以能够得到核函数对象的特化版本，是因为早在运行时核心注册核函数子类时，同一核函数子类的多个特化实现已经被一组 `REGISTER_KERNEL_BUILDER` 宏一并注册到了 `KernelRegistration` 对象中。`create_kernel` 指向的匿名函数内部通过一系列带有缓存机制的工具类，级联调用定义于 tensorflow/core/framework/op_kernel.cc 文件的 `CreateOpKernel` 全局函数。该函数使用带有设备类型参数的 `OpKernelConstruction` 对象，调用 `KernelRegistration` 对象关联的核函数工厂方法，完成核函数特化实例的创建。

对于矩阵乘法操作，它的核函数定义于 tensorflow/core/kernels/matmul_op.cc 文件。`MatMulOp` 类模板的签名为 `template <typename Device, typename T, bool USE_CUBLAS> class MatMulOp : public OpKernel`，其中 `Device`、`T` 和 `USE_CUBLAS` 参数分别指定实例化的类对应的设备类型、元素数据类型和 cuBLAS 支持性。这个类模板具有 `Device` 参数为 `CPUDevice`、`GPUDevice` 和 `SYCLDevice` 的实例化类型，其辅助数据结构 `MatMulFunctor` 和 `LaunchMatMul` 具有针对不同设备类型的特化实现，因此可以认为矩阵乘法操作具有针对 3 种设备的特化版本。

名称为“[操作名]+Functor”和“Launch+[操作名]”的辅助数据结构并非 MatMulOp 类所独有，TensorFlow 的很多计算类操作的核函数均采用这一设计惯例。二者都是封装核函数实现逻辑的结构模板，前者一般具有重载括号运算符的 operator()方法，后者一般具有名为 launch 的公开方法。它们通常带有表示设备类型、数据类型和其他操作属性的模板参数，并对不同设备进行了特化实现。二者的嵌套使用能够为核函数模板的实例化带来一定的灵活性。

矩阵乘法操作的代表性，体现在它使用 Eigen 库实现面向 CPU 和 OpenCL GPU 的核函数，使用 cuBLAS 或 NVIDIA 的其他开发库实现面向 CUDA GPU 的核函数——这是 TensorFlow 的多数代数算法操作选择的实现路径。下面分别以 CPU 和 CUDA GPU 为例说明其具体流程。

14.5.2　CPU上的执行流程

图 14-14 给出了矩阵乘法操作在 CPU 上执行的主要流程。该操作的核函数子类 MatMulOp 以 CPUDevice 作为设备类型的实例化参数，兼容浮点数和整数等元素数据类型，且不开启 cuBLAS 支持。辅助结构 LaunchMatMul 的特化版本依次继承了 LaunchMatMulCPU 和 LaunchMatMulBase 父类，这是为了同 OpenCL GPU 的核函数实现代码复用。

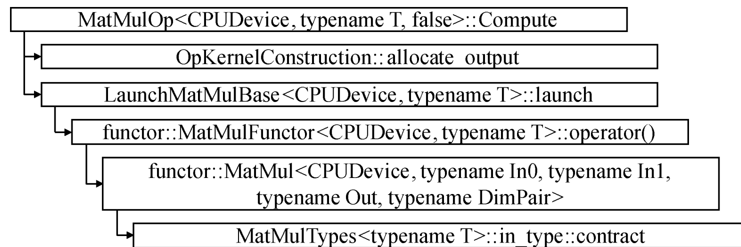


图 14-14　MatMulOp 在 CPU 上执行的主要流程

会话执行器通过 ExecutorState::Process 方法执行核函数时，MatMulOp::Compute 方法将被设备对象的 Compute 方法调用。MatMulOp::Compute 方法首先基于输入张量的形状信息为输出分配内存空间，然后通过 LaunchMatMulBase::launch 及其级联调用的 MatMulFunctor::operator() 方法启动计算逻辑的执行。基于 Eigen 库实现的计算逻辑包装在名为 MatMul 的全局函数中，核心语句为：out.device(d) = in0.contract(in1, dim_pair);。其中，输入参数的类型为 MatMulTypes<typename T>::in_type。这是一个类型别名，其实际类型是 Eigen::TensorMap< Eigen::Tensor<const T, 2, Eigen::RowMajor>, Eigen::Aligned>，即元素数据类型为 T 的二维张量。Eigen::TensorMap 类的 contract 方法实现了张量缩并（contraction）运算，在这一特例场景下等价于矩阵乘法。

14.5.3　CUDA GPU上的执行流程

图 14-15 给出了矩阵乘法操作在 CUDA GPU 上执行的主要流程。该操作的核函数子类 MatMulOp

以 `GPUDevice` 作为设备类型的实例化参数，兼容浮点数和整数等元素数据类型，默认开启 cuBLAS 支持。辅助结构 `LaunchMatMul` 的特化版本借助 StreamExecutor 库调用 cuBLAS 库提供的算法。

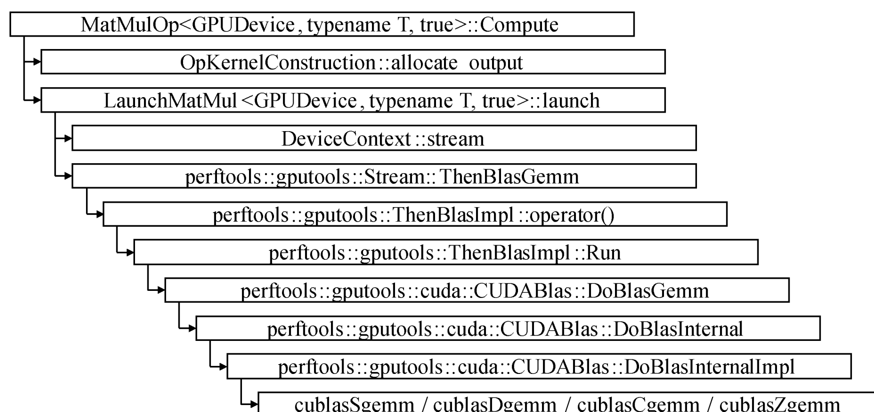


图 14-15 `MatMulOp` 在 CUDA GPU 上执行的主要流程

`MatMulOp` 在 CUDA GPU 上的入口逻辑与 CPU 的情况类似。特化的 `LaunchMatMul::launch` 函数首先需要从 StreamExecutor 库取得在 GPU 上调用执行函数所需的流对象。接下来，依据输入矩阵的形状选择不同的调用路径。对于两个输入矩阵均非一维向量的一般情况，调用流对象的 `perftools::gputools::Stream::ThenBlasGemm` 方法执行乘法操作。为了实现从 C++ 重载方法到 cuBLAS 不同参数类型函数的映射，同时实现异构内存管理、指针类型转换和日志等功能，StreamExecutor 内部对 cuBLAS 函数调用进行了多层封装。这些封装函数最终会依据输入矩阵的元素数据类型，从 `cublas*gemm` 系列矩阵乘法函数中选择适当的一个加载到流上执行。例如，对于单精度浮点数矩阵，使用 `cublasSgemm` 函数计算。

对于其中一个输入矩阵为一维向量的情况，`LaunchMatMul::launch` 函数会调用 `LaunchBlas-Gemv::Compute` 方法进入优化路径。这一路径最终使用 cuBLAS 的 `cublas*gemv` 系列函数，执行矩阵–向量乘法运算。

`MatMulOp` 在 OpenCL GPU 上执行的主要流程同 CPU 类似，其差别在于 Eigen 库的内部实现，这里不再赘述。

14.6 小结

数据流图计算是 TensorFlow 最为核心的工作。这一机制作为运行时库的“引擎”，驱动着各个组件有序运转。围绕控制流与数据流两条主线，TensorFlow 核心层提出了 Session、Graph 等一系列抽象，设计了层次化的数据流图计算调用栈。针对数据流图这一中心抽象，TensorFlow 具有灵活而高效的数据结构与控制逻辑，能够实现全图构造、子图提取、面向设备的图切分以及面向执行的图优化。TensorFlow 使用会话机制管理计算流程及其相关对象的生命周期。对于单机会话，

它为会话生命周期整体、会话单次运行和会话在单个设备上的执行提供不同粒度的执行主体与状态管理抽象，将计算过程组织为可以流水线式并发执行的多个阶段，旨在提高系统效率与资源利用率。对于分布式会话，它提供一种主–从模型，将会话生命周期分解到不同角色的组件上，为应用层 API 的灵活性和算法模型的可伸缩性提供支撑。数据流图的计算逻辑最终通过操作核函数封装的 Eigen 和 cuBLAS 等底层算法库落实到了具体的计算设备。理解数据流图的计算原理是掌握 TensorFlow 核心机理的关键，也是进行运行时库二次开发的必由之路。

第五部分

生态发展篇

- 第 15 章　TensorFlow 生态环境



第 15 章

TensorFlow 生态环境

生态系统是开源软件获取生命力的主要源泉。Linux、Hadoop 等堪称经典的开源软件很大程度上依靠着生态环境的友好和开放促成了其发展与成功。TensorFlow 社区自软件开源之初便秉持着开放的态度，一方面积极吸纳第三方开发者贡献的代码和设计思想，另一方面也热切欢迎第三方系统适配或集成 TensorFlow。在开源短短两年的时间内，TensorFlow 周边项目大量涌现——从算法模型到辅助工具，从易用性封装到软件间适配，乃至为之打造的专用硬件——这预示着 TensorFlow 将会成为开源软件界的又一个经典。本章选取 TensorFlow 生态系统中的若干代表性项目进行介绍。在此基础上，对 TensorFlow 社区的未来发展加以展望。

15.1　生态环境概况

TensorFlow 社区非常重视软件生态环境建设，其开源理念与 Linux 和 Hadoop 社区类似，即社区的管理者和贡献者专注于维护 TensorFlow 核心组件，着力将其打造为机器学习、深度学习领域通用而强大的计算引擎，同时对第三方贡献的重要特性进行选择性、渐进式的集成。在此基础上，鼓励第三方开展 TensorFlow 的产品化封装、易用性设计、算法模型研发，以及与特定应用需求或专用硬件环境相关的定制化开发。在这一理念的指导下，TensorFlow 社区自身托管着若干生态系统组件，同时为他人开发相关项目提供技术支持。社区托管的组件可分 3 类：TensorFlow 源代码包 contrib 目录中的组件、ecosystem 项目中的组件，以及 TensorFlow 团队管理的其他组件。第三方项目既包括 TensorFlow 的众多复刻（fork）和衍生版本，也包括针对市场需求或行业特征而开发的周边软硬件产品。下面简单介绍几类生态系统组件的特点及典型代表。

15.1.1　社区托管组件

TensorFlow 社区托管组件是指以 TensorFlow 开发团队（用户 ID：tensorflow）的名义在 GitHub 上开源的周边组件。这些组件有些来自 Google 公司，有些来自第三方开发者。它们依据用途和成熟度的差异，被放置在不同的项目和目录中。

1. contrib 组件

TensorFlow 源代码包的 tensorflow/contrib 目录汇集了开源贡献者开发的数十种组件。这些组

件通常具有较高的实用性和通用性，通过了社区管理者的审核及自动化构建系统的验证，有潜力成为 TensorFlow 核心组件的一部分。但可能由于组件的技术成熟度不够高、兼容性受限或其他特殊原因，暂时不能进入 TensorFlow 核心代码。社区期望广大用户对这些组件进行充分的测试和讨论，然后决定其未来地位。由于开发者会根据社区和用户的反馈对组件进行及时修改，这类组件的 API 稳定性不如核心组件，在生产系统中使用时需要特别留意。contrib 组件涉及的特性广泛，可以大致分为算法与模型、系统平台支持、专用硬件支持、编译构建脚本和实用工具等几类。典型组件介绍如下。

- **算法与模型**

- distributions：若干统计分布算法的集合，用于构建基于概率分布的模型。
- layers：层次化的机器学习组件集合，包括构建模型常用的卷积层、池化层、优化器、正则化函数等组件。引入粗粒度的操作，有利于用户快速开发复杂的模型。
- learn：深度学习及机器学习模型训练工具类的集合。提供高层次 API，以便数据科学应用的开发者快速搭建、评价和调优模型。其中包含用于搭建模型实验环境的 `Experiment` 和 `LearnRunner` 抽象，以及模型评价器（estimator）和训练监督器（monitor）等工具。针对多种深度学习及机器学习算法的模型训练需求实现了相应的子类。

- **系统平台支持**

- android：针对 Android 移动平台提供支持的组件，主要包含一套基于 Java 语言和 JNI 机制开发的 TensorFlow API 代码。与之对应地，tensorflow/examples/android 目录给出了一组 Android 版 TensorFlow 应用示例，均为基于神经网络模型预测算法的图形图像应用。
- ios_examples：面向 iOS 移动平台的示例应用代码。由于 iOS 平台上的 Objective-C++ 语言能够方便地链接 C++ 库，TensorFlow 不需要为 iOS 提供专门的适配层 API。
- pi_examples：面向树莓派（Raspberry Pi）开发板的示例应用代码。树莓派本质上是一套运行 Linux 操作系统的 ARM 架构计算机，同一般的 Linux 环境无本质区别，因此 TensorFlow 不需要为之提供专门的适配层 API。

- **专用硬件支持**

- hvx：Hexagon Vector eXtensions（HVX）支持组件。HVX 是高通公司 Hexagon DSP 芯片的向量计算加速特性，主要用于移动设备上的图像处理等计算逻辑加速。TensorFlow 的 Android 版本可以利用这一组件提升性能。
- verbs：针对高性能计算环境中常用的 InfiniBand 网卡及 RDMA 协议的通信模块。在具有这类高端网络硬件的集群环境下，该模块能够发挥网络带宽潜力，提升 TensorFlow 分布式运行模式的数据通信性能。

- **编译构建脚本**

- cmake：基于 CMake 工具链的构建脚本，主要用于 Windows 系统下的 TensorFlow 编译构建。

- ❑ makefile：基于 GNU Make 工具链的构建脚本，主要适用于编译目标平台为 Android、iOS 等系统的场景。在这类情况下，Bazel 的特性不能完全满足 TensorFlow 的编译构建需求。

- **实用工具**

- ❑ imperative：命令式编程封装模块。该模块将 TensorFlow API 以命令式编程模式导出，允许用户编写带有传统控制结构、能够实时访问中间结果的应用程序。它适用于交互式调试，以及与其他命令式开发库协同使用的场景。
- ❑ tfprof：TensorFlow 性能分析器。它能够以多种不同的视图剖析应用程序中的操作执行时间和内存占用情况，呈现检查点中的张量数据，从而帮助开发者理解代码的运行时行为，服务于模型参数调优、程序性能优化等场景。

2. ecosystem 组件

TensorFlow 团队开放的 ecosystem 项目（https://github.com/tensorflow/ecosystem）包含用于构建 TensorFlow 与周边生态系统交互关系的组件。这些组件与 contrib 组件的显著差别在于：其代码并不运行在 TensorFlow 环境中，而是运行在周边生态系统的软件环境中。ecosystem 项目目前提供的组件分为两类：运行时环境支持组件和外部数据结构支持组件。前者适用于在云计算、虚拟化环境中自动化部署 TensorFlow 集群，后者服务于大数据环境下不同计算引擎之间的数据交换。现有组件介绍如下。

- **运行时环境支持组件**

- ❑ docker：TensorFlow 的 Docker 容器配置文件，包括在容器内安装 HDFS 的脚本。用来基于 Docker 镜像快捷部署独立的 TensorFlow 运行环境。
- ❑ kubernetes：针对 TensorFlow 作业的 Jinja 模板文件，用于在 Kubernetes 集群环境中以容器方式启动和管理 TensorFlow 作业。
- ❑ marathon：针对 TensorFlow 作业的另一种 Jinja 模板文件，用于在 Marathon（DC/OS）集群环境中以容器方式启动和管理 TensorFlow 作业。

- **外部数据结构支持组件**

- ❑ hadoop：该模块是一种适用于 Hadoop 和 Spark 的 Java 开发库，其功能是将 TensorFlow 的 TFRecords 数据格式作为 Hadoop 和 Spark 的输入/输出数据格式（`InputFormat/OutputFormat`）使用。
- ❑ spark：该模块实现了 TensorFlow TFRecords 与 Spark DataFrame 两种数据格式的相互转换功能。它使用 Scala 语言编写，可以在 Spark Shell 或 Spark 应用程序中使用。

3. TensorFlow 团队的其他项目

除了 TensorFlow 和 ecosystem 外，TensorFlow 团队还在 GitHub 上开放了其他一些项目。这些项目大多来自 Google 公司，是 TensorFlow 官方开发者贡献的周边项目，涉及算法模型集（models、tensor2tensor）、服务化工具（serving）、动态图计算库（fold）、基准测试框架（benchmarks）、

可视化平台（playground）、编程语言接口（haskell、rust）、研究型应用（deepmath）和创意类应用（magenta）等。部分项目已在之前章节中说过，这里简单介绍其他几个有特色的项目。

- benchmarks：TensorFlow 的基准测试框架，提供多种经典 CNN 模型的参考实现代码，以及基于 Kubernetes 的容器化分布式测试脚本，用于收集、呈现和分析深度学习平台的运行时性能。
- fold：该项目用于在 TensorFlow 平台上实现动态图计算。它基于 Dynamic Batching 算法，能够使用 TensorFlow 固有的静态图计算引擎高效地模拟动态图计算语义，从而提供灵活的模型构建能力。诸如 TreeLSTM 等数据流图构建过程依赖于输入数据的模型可以通过 Fold 完美实现。
- magenta：一套基于神经网络的人工智能作曲系统，它展示了 TensorFlow 在艺术创作方面的应用前景。
- playground：基于 Web 的深度学习交互式可视化展示平台，用于向初学者直观演示神经网络模型的工作原理。读者亦可直接访问 TensorFlow 官网部署的 playground 环境——http://playground.tensorflow.org。
- tensor2tensor：Google Brain 团队推出的模块化、可扩展的深度学习库，包含 Transformer、SliceNet 等多种新近发布的网络模型，以及搭建训练系统所需的若干工具组件。该项目适用于机器翻译、图像信息描述等多个应用领域，有助于工业界快速应用 Google 的研究成果。

Google 公司对 TensorFlow 开源生态系统的贡献为第三方开发者做出了表率。关注 Google 公司，特别是 TensorFlow 团队的新兴开源项目，是了解机器学习、深度学习领域发展动向的渠道之一。

15.1.2　第三方项目

以 Apache License 开源的 TensorFlow 在商业使用和二次发布方面的限制很少，因此第三方复刻和衍生开发的版本众多，这些版本通常会对 TensorFlow 进行针对性的改进。同时，第三方为满足自身或市场需求而开发的周边项目也是 TensorFlow 生态环境的重要组成部分。这些项目不仅包含软件产品，也包含可与 TensorFlow 协同工作的硬件产品。

在商业公司方面，TensorFlow 官方网站提供了一组使用 TensorFlow 的公司列表。事实上，这不仅是一个用户列表，这些榜上有名的公司或多或少也为 TensorFlow 生态系统建设作过贡献。例如，eBay 公司开源了基于 TensorFlow 的 Sequence-Semantic-Embedding 自然语言处理工具集，IBM 的 PowerAI 机器学习系统将 TensorFlow 移植到了 Power 架构服务器，Intel 的神经网络处理器能够为 TensorFlow 平台提供加速支持，小米科技的工程师也为 TensorFlow 贡献过代码和应用案例。

在开源社区方面，TensorFlow 官方并未专门提供第三方项目列表，不过有不少热心人士汇总

过 TensorFlow 相关项目的信息，其中较为有名的是 Awesome TensorFlow 项目（https://github.com/jtoy/awesome-tensorflow）。该页面中的 Models/Projects、Libraries 分类收集了若干活跃度较高的第三方项目，读者可以从中检索自己感兴趣的关键词。本章后面将介绍其中几个具有代表性的软硬件项目。

15.2　深度神经网络库 Keras

Keras（https://keras.io）是一套抽象层次高且极度模块化的深度神经网络 API。它的设计目标是尽量减少从一个好想法到实验结果的时间，旨在为算法工程师或研究人员提供快速验证模型效果的能力。Keras 由 Python 语言编写而成，它提供一套经过统一封装的 API，并以 TensorFlow 或 Theano 作为后端的计算引擎。自 2015 年创立项目至今，从用户数量和关注度的角度看，Keras 已成为全球最受欢迎的深度学习开源项目之一。

15.2.1　概述

原生 TensorFlow API 以灵活性和多样性见长，但是对于算法工程师或研究人员来说，其学习成本和编程门槛较高。而 Keras 以其用户友好和快速的原型设计能力吸引了大量用户。它提供 CNN 和 RNN 等网络，能够无缝运行在 CPU 和 GPU 设备上。同时，其 API 设计更加简单易用，兼容 Python 2.7 至 Python 3.6 的所有版本。

相比 TensorFlow 原生 API，Keras 具有以下优点。

- **用户友好**：Keras 将用户体验放到首位，在参考了大量深度学习库的最佳实践和常见用例的基础上，总结出一套简洁而统一的 API，最大可能地减少用户的学习成本和编程门槛。同时，Keras 也提供了清晰和简便的 bug 反馈机制，利用社区的力量修正错误和改进不足。
- **模块化**：深度神经网络模型可以简单理解为一系列神经网络的有序堆叠或一幅具有拓扑结构的数据流图。Keras 内置了大量神经网络模块，包括但不限于卷积层、池化层、循环层、嵌入层、激活层、正则层等。用户通过模块的自由组合与堆叠，便可快速构建出自己的深度神经网络模型。
- **扩展方便**：在 Keras 中自定义新模块是一件非常简单的工作，只需要参考现有的模块设计新的类或函数即可。这使得算法研究人员能够基于 Keras 便捷地开展更加先进和个性化的研究。
- **编程简单**：Keras 完美支持 Python 语言。用户无需额外导入模型配置文件，直接使用 Python 即可完成算法模型的开发和微调等工作。

Keras 的主要优势是提供了面向对象的编程接口，有效缓解了用户编写神经网络模型的复杂度，降低了深度学习的编程门槛，加速了深度学习应用的落地。但是，Keras 与 TensorFlow 原生 API 相比也有不足，主要体现在以下几点。

- **不支持分布式模型**：Keras 为了解耦后端引擎，自己定义了一套优化器，如 SGD、RMSprop 等。因为它没有继承和封装 TensorFlow 的优化器，所以无法使用 TensorFlow 为分布式模型设计的同步优化器。这导致用户无法使用 Keras 编写分布式模型。这一问题涉及 TensorFlow 底层的分布式设计，Keras 短期内无法解决。
- **模块通用性不足**：Keras模块化设计的通用性较差，只能设计神经网络模型，无法编写传统的机器学习模型，如 Logistic 回归、K-means、随机森林、支持向量机、高斯混合模型等。相比之下，TensorFlow 的 contrib 目录下早已接受了大量传统机器学习模型的代码。不过，近期已经有人开始尝试使用 Keras 进行强化学习的研究，Keras 未来的发展仍值得期待。
- **编程语言支持有限**：Keras在多语言API的支持上还不够广泛，目前仅支持Python语言。相比之下，TensorFlow 官方已经发布了 Python、C++、Go 和 Java 等语言的 API。

从 TensorFlow 1.2 开始，Keras 合入了 contrib 目录，成为 TensorFlow 发行版的一部分。这为用户带来两个好处：其一，用户再也不用分别安装 TensorFlow 和 Keras 软件包；其二，用户不用担心两者的版本适配问题。毋庸置疑，这对于 TensorFlow 和 Keras 的用户来说都是利好消息。这一决定将会进一步扩大两者的用户数量，加深两个社区的互动，补齐 TensorFlow 原生 API 难上手的短板。同时，这也表明 TensorFlow 团队完全肯定了 Keras 在 TensorFlow 生态圈建设上的贡献。

15.2.2　模型概述

Keras 提供两种模型范式，顺序（sequential）模型和函数式模型（model/functional API）。

- **顺序模型**：一系列神经网络层的线性堆叠，适用于单输出的线性神经网络模型。它是函数式模型的特殊形式。
- **函数式模型**：符合用户输入输出定义且极易重用的模型，适用于多输入多输出神经网络模型。

因为顺序模型是函数式模型的特殊形式，其实现也基于函数式模型的抽象，所以两种模型具有一些公共方法，如表 15-1 所示。

表 15-1　Keras 两种模型的公共方法

方法名称	功能说明
`model.summary`	打印模型汇总信息
`model.get_config`	返回包含模型配置信息的 Python 字典
`model.get_layer`	根据名称或索引获取网络层对象
`model.get_weights`	返回模型权重值，数据类型为 NumPy 数组
`model.set_weights`	根据 NumPy 数组，设置模型权重值，两者的形状必须相同
`model.to_json`	返回仅包含模型网络结构的 JSON 字符串，利用该字符串可重构模型
`model.to_yaml`	返回仅包含模型网络结构的 YAML 字符串，利用该字符串可重构模型
`model.save_weights`	将模型权重值保存到指定路径，保存的文件类型为 HDF5
`model.load_weights`	从指定 HDF5 文件加载模型权重值到当前模型

注：表中的 `model` 表示顺序模型和函数式模型的实例。

Keras 模型的典型使用步骤为：

(1) 定义神经网络结构，创建模型；

(2) 配置优化器、损失函数和度量指标，编译模型；

(3) 填充数据和标签，训练模型。

15.2.3　顺序模型

Keras 顺序模型是一系列神经网络层的线性堆叠，适用于单输出的线性神经网络模型。它是函数式模型的特殊形式。利用 Keras 内置的神经网络层，用户可以快速创建和训练顺序模型。如果想要使用 TensorFlow 原生 API 实现相同的功能，其编程复杂度可能更高。

下面我们详细介绍顺序模型的 3 个典型使用步骤：

(1) 创建模型；

(2) 编译模型；

(3) 训练模型。

1. 创建模型

keras.models.Sequential 是 Keras 的顺序模型类，它的构造方法接受的输入参数是表示模型网络结构的列表。表中元素保存的是有序的、任意类型的网络层，用户需要确保各网络层之间输入和输出张量的形状合法。Keras 提供两种创建顺序模型的方法：一种是在创建顺序模型实例时，通过输入参数指定所有网络层；另一种是先实例化顺序模型，然后依次添加各网络层。下面我们以两层全连接网络为例，分别使用这两种方法创建模型。

方法一：首先，从 keras.models 模块导入顺序模型类 Sequential。然后，从 keras.layers 模块导入全连接层 Dense 和激活层 Activation。接着，使用 Sequential 类创建顺序模型实例 model，并设置其为两层全连接网络。同时，分别设置两个全连接层的激活函数为 relu 和 softmax。代码如下所示：

```
from keras.models import Sequential
from keras.layers import Dense, Activation
# 创建模型时指定所有网络层
model = Sequential([
    Dense(32, units=784),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

方法二：首先，导入相应的类。然后，创建顺序模型实例 model。接着，调用 model.add 方法依次添加各层网络。最后，成功创建出与方法一同样的模型。代码如下所示：

```
from keras.models import Sequential
from keras.layers import Dense, Activation
# 用 model.add 方法创建方法一定义的顺序模型
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))
```

2. 编译模型

用户需要调用 model.compile 方法为模型配置优化器、损失函数和度量指标。其中，优化器和损失函数是编译模型必需的参数，度量指标是可选参数。用户可以使用 Keras 内置的优化器、损失函数和度量指标，也可以根据实际需求自定义。如果想要自定义度量指标，那么需要创建对应的指标评估方法。下面我们以典型的二分类和多分类问题为例，展示编译 Keras 顺序模型的方法：

```
import keras.backend as K

def correct_prediction(y_label, y_prediction):
  return K.mean(y_prediction)
# 二分类问题，使用自定义的 correct_prediction 评估方法
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', correct_prediction])
# 二分类问题，使用内置的准确率作为度量指标
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
# 多分类问题，使用内置的准确率作为度量指标
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

3. 训练模型

用户需要调用 model.fit 方法训练模型。其中，填充的数据和标签均为 numpy 数组类型。下面的代码展示了二分类模型的训练流程：

```
# 创建顺序模型
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
# 编译模型，设置训练指标为准确率（预定义指标）
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
# 生成哑（dummy）数据
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))
# 设置 epochs 为 10，批数据大小为 32，开始训练
model.fit(data, labels, epochs=10, batch_size=32)
```

综上，使用 Keras 训练模型的过程比使用 TensorFlow 原生 API 更加简单。Keras 封装了模型以外的绝大多数编程接口，使得用户更加专注于模型本身的设计和训练过程。

15.2.4　函数式模型

`keras.models.Model` 是 Keras 的函数式模型类。用户只要指定模型的输入和输出，即可快速创建函数式模型。同时，这个类还支持多输入和多输出。因此，相比顺序模型，它的灵活性和通用性更好。`Model` 的构造方法接受两个输入参数——`inputs` 和 `outputs`，分别表示模型的输入和输出。我们可以使用以下代码创建一个单输入单输出的函数式模型：

```
from keras.models import Model
from keras.layers import Input, Dense
a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

如果想要创建多输入多输出的函数式模型，就需要传入相应的输入输出列表。例如：

```
model = Model(inputs=[a1, a2], outputs=[b1, b3, b3])
```

函数式模型带来的好处主要有两点：方便复用模型、灵活构建模型。下面我们通过代码示例，在介绍函数式模型使用流程的同时对这两个好处进行说明。

1. 方便复用模型

当用户调用预训练的模型时，重用的不仅是模型的网络结构，而且包括模型的参数值。因此，用户只需要传入符合模型定义的输入数据，就可以得到符合预期的输出结果，这极大地方便了模型的微调和二次开发。

下面的代码实现了解决图像分类问题的全连接网络。首先，我们定义模型的输入 `inputs` 和输出 `predictions`。然后，编译模型，并设置优化器为 `rmsprop`。接下来，填充数据和标签，开始训练模型。

```
from keras.layers import Input, Dense
from keras.models import Model

inputs = Input(shape=(784,))

x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)
```

假设经过多轮迭代训练之后，我们已经得到一个准确率较高的图像分类模型。现在，如果需

要创建一个视频分类模型，就可以复用前一模型。事实上，只要对图像分类模型稍作修改，即可得到视频分类模型。下面的代码能够处理一段包含 1000 帧图像的视频，这一模型将会输出每一帧图像所属的类别：

```
from keras.layers import TimeDistributed
# 设置输入张量形状为(1000, 784)
input_sequences = Input(shape=(1000, 784))
# 将图像分类模型应用到输入队列中的每一张图片，使其能够处理视频分类问题
processed_sequences = TimeDistributed(model)(input_sequences)
```

2. 灵活构建模型

函数式模型的设计方便用户快速灵活地构建模型，这使得处理大量错综复杂的数据流成为可能。

假设我们要设计一个预测 Twitter 推文被转发和点赞次数的模型。它的主要输入是推文标题，即一个词向量；辅助输入是推文发布日期等附加信息。那么，模型的损失函数将由两部分组成：主损失函数——评估基于推文标题和发布日期等辅助信息的预测结果；辅助损失函数——评估仅基于推文标题本身的预测结果。辅助损失函数旨在摆脱主损失值的干扰，使我们能够平滑地训练嵌入层和 LSTM 层的模型参数。对于深度神经网络模型来说，尽早使用主损失函数有利于模型的正则化。

该模型的拓扑结构如图 15-1 所示。首先，输入文本格式的推文。然后，添加嵌入层，目的是用词向量嵌入表示对应的自然语言。接着，加入 LSTM 网络层，期望在向量空间中学到表达能力更强、更准确的语言模型，使得词向量能够尽可能还原对应的推文标题。最后，合并推文标题和辅助输入，基于主损失值和辅助损失值的加权结果，使用一个全连接层拟合预测函数，并得到预测结果。

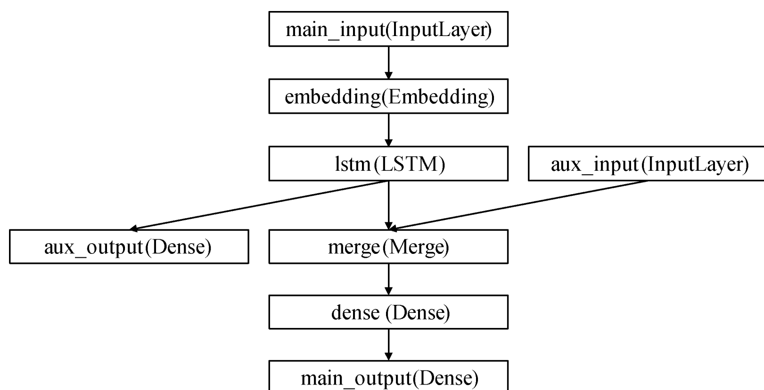


图 15-1 预测 Twitter 推文转发和点赞次数模型的拓扑结构

下面我们用 Keras 来实现如图 15-1 所示的预测模型。该模型的主要输入（`main_input`）是推文标题，即长度不超过 100 的 One-Hot 编码整数序列。其中，每个整数的取值范围都是 [1,10000]。换句话说，单词表的词汇数为 10000。将 One-Hot 编码的推文标题序列输入嵌入层重新编码，输出 512 维的稠密词向量（`x`）。

```
from keras.layers import Input, Embedding, LSTM, Dense
from keras.models import Model

main_input = Input(shape=(100,), dtype='int32', name='main_input')
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)
```

将嵌入层输出的词向量传入 LSTM 层，学习能够表达整个推文标题语义的词向量（`lstm_out`）。通过优化辅助损失函数，独立地训练嵌入层和 LSTM 层的参数；通过 logistic 回归层得到辅助输出（`aux_output`）。相关代码如下：

```
lstm_out = LSTM(32)(x)
aux_output = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)
```

然后，将辅助输入（`aux_input`）的推文发布日期和 LSTM 层输出的词向量（`lstm_out`）连接起来，组成完整的输入词向量（`x`）。将其输入一个全连接层，最终同样通过 logistic 回归得到主要输出（`main_output`）。相关代码如下：

```
aux_input = Input(shape=(5,), name='aux_input')
x = keras.layers.concatenate([lstm_out, auxiliary_input])
x = Dense(64, activation='relu')(x)
main_output = Dense(1, activation='sigmoid', name='main_output')(x)
```

至此，我们已经创建了该神经网络模型的所有模块。层与层之间的张量形状由 Keras 自动推断，用户只需定义清楚输入和输出即可创建该模型的实例：

```
model = Model(inputs=[main_input, aux_input], outputs=[main_output, aux_output])
```

编译模型时，我们通过 `loss_weights` 参数将主损失值和辅助损失值分别初始化为 1.0 和 0.2。同时，这里只传入了一个损失函数——`binary_crossentropy`，这表明我们的主损失函数和辅助损失函数都将使用 `binary_crossentropy`。相关代码如下：

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', loss_weights=[1., 0.2])
```

最后，填充训练数据和标签，开始训练模型：

```
model.fit([headline_data, additional_data], [labels, labels], epochs=50, batch_size=32)
```

综上可以看出，使用 Keras 的函数式模型定义多输入多输出的复杂神经网络模型是非常便捷和灵活的。利用 Keras 内置的大量网络层、激活函数、损失函数、优化器等模块，可以大幅提升算法工程师和研究人员验证原型的效率。

15.3　TensorFlow 与 Kubernetes 生态的结合

Kubernetes（https://kubernetes.io）是 Google 公司于 2014 年开源的容器集群管理系统。它继承了 Google 公司多年来在生产环境中管理大规模容器应用的经验，发扬了 Google 内部容器管理系统 Borg 的设计理念和先进思想。Kubernetes 一经推出，便受到了全球云计算领域从业者的广泛欢迎。时至今日，Kubernetes 已经成为容器生态系统的重要组件和容器编排领域的事实标准。其推崇的像“牲畜”而非“宠物”一样管理服务的理念，如今也成为微服务架构设计的指导思想。

TensorFlow 作为 Google 公司在人工智能领域的拳头产品，除了满足全球算法工程师的科研需求外，还需要考虑在生产环境中解决实际问题。从实验室原型到商用产品，TensorFlow 自身还存在两点不足：其一，直接使用 TensorFlow Serving 挂载模型提供推理服务的方式无法应对高并发访问。人工添加负载均衡的成本较高。访问流量降低时无法迅速回收资源，导致资源利用率降低。其二，分布式训练作业需要用户在每台机器上手动启动 TensorFlow 进程，并且显式指定每一台机器的主机地址和端口号。当 TensorFlow 集群规模增大时，这种方式的易用性差，用户体验十分不友好。

如果能够将 TensorFlow 作业运行在容器中，利用 Kubernetes 的容器编排能力管理 TensorFlow 作业，就可以显著改善这两点不足。下面我们就以 TensorFlow 推理服务和分布式训练作业为例，介绍基于 Kubernetes 的解决方案。

1. 使用 Kubernetes 管理 TensorFlow 推理服务

我们认为 TensorFlow 的推理服务与长周期的 Web 服务类似。在引入容器管理方案时，需要重点考查方案提供的可用性和可伸缩性是否能够达到生产环境的需求。在 Kubernetes 的世界里，一切服务都运行在容器中。最简单的容器组抽象是 Pod，一个 Pod 中支持同时运行多个容器。

Kubernetes 的 Service 抽象提供反向代理的能力，它负责将网络上的用户请求转发给 Pod 中运行的服务，并将服务的响应返回给网络上的用户。在 TensorFlow 推理服务的用例中，Service 管理的服务就是 TensorFlow Serving。Service 通过识别不同 Pod 的标签（容器名称、类型和模型名称等）来筛选提供对应服务的 Pod。同时，它也提供简单的负载均衡能力。当有多个 Pod 副本提供相同服务时，Service 会对用户的请求进行负载分发，以确保所有请求都能够得到快速响应。当请求数量较大导致单个 Service 无法处理时，Kubernetes 支持在内部运行独立的 Nginx 容器以提供负载均衡服务。当请求的数量上升时，为了能够持续提供稳定的服务，需要增加后台服务的数量。只要服务本身是无状态的（TensorFlow Serving 即属于这种情况），在使用 Kuberentes 实现可伸缩性时就不需要对服务做任何修改。

如果我们需要在服务运行时增减 Service 代理的后台 Pod 数量，那么可以选择使用 Kubernetes 的高级抽象——ReplicaSet 和 Deployment。ReplicaSet 是相同 Pod 的副本集合，Deployment 为 ReplicaSet 的升级和管理提供了一套描述语法。用户只需要使用 Kubernetes 的命令行工具 `kubectl` 描述期望服务达到的状态，Kubernetes 在内部就可以完成资源的调度工作。因此，利用 ReplicaSet 和 Deployment 可以实现服务的弹性伸缩。

图 15-2 展示了使用 Kubernetes 管理 TensorFlow 推理服务的一个实例。我们在 Kubernetes 集群中部署了两套 TensorFlow 推理服务 A 和 B，分别由 Service A 和 Service B 代理用户请求。Service A 负责将网络上的用户请求转发给它代理的两个 Pod，这两个 Pod 均属于 ReplicasSet A。如果服务 A 的请求数量上升，我们可以调用 `kubectl scale deployment replicaset-a --replicas=10` 命令将副本数量扩容为 10 个。类似地，我们也可以完成服务的缩容操作。不难发现，使用 Kubernetes 管理 TensorFlow 推理服务是非常方便的。

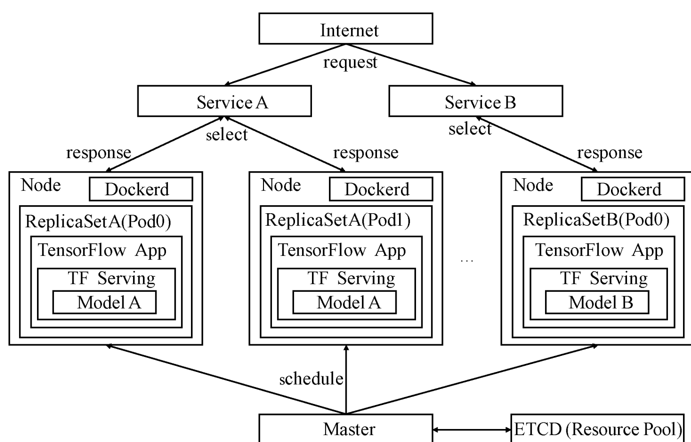


图 15-2　TensorFlow 推理服务在 Kubernetes 集群中的逻辑部署图

2. 使用 Kubernetes 管理 TensorFlow 分布式训练作业

TensorFlow 分布式训练作业的特征与高性能计算或大数据分析系统中的批处理作业类似。TensorFlow 分布式训练作业由多个 worker 任务和多个 PS 任务组成。worker 任务负责模型参数和对应梯度的计算，PS 任务负责模型参数的存储、分发和更新。TensorFlow 分布式训练作业在实际使用时有以下 4 个痛点。

- ❑ 缺少统一资源调度。
- ❑ 缺少统一任务启动。
- ❑ 手动指定任务地址。
- ❑ 缺少统一资源回收。

针对 TensorFlow 分布式训练作业难使用和难管理的问题，TensorFlow 团队的 ecosystem 项目（https://github.com/tensorflow/ecosystem）给出了一个相对合理的解决方案。下面我们略微改进此方案，介绍如何使用 Kubernetes 解决上述 4 个痛点。

- ❑ Kubernetes 作为容器编排系统，本身就带有资源调度的功能。因此，如果在 Kubernetes 集群中部署和运行 TensorFlow 分布式训练作业，那么天然享受 Kubernetes 的统一资源调度能力。
- ❑ Kubernetes 的作业描述文件格式为 YAML（YAML Ain't Markup Language），这是一种适用于所有编程语言的数据序列化格式。我们知道，worker 任务和 PS 任务仅通过启动参数区分作业类型和任务编号。如果想要统一启动一组作业的所有任务，那么可以使用 Jinja 模板引擎渲染各个任务对应的 YAML 文件。这些 YAML 文件中的差异项满足一定规则，这些规则可以通过 Jinja 模板引擎的语法进行描述。读者可以参考 ecosystem 项目给出的 Jinja 模板（https://github.com/tensorflow/ecosystem/blob/master/kubernetes/template.yaml.jinja）。该 Jinja 模板生成的 YAML 文件可经由 Kubernetes 调用，用于启动 TensorFlow 分布式训练作业。

- 统一启动所有的worker任务和PS任务后，我们还需要确保它们能够相互通信。因为我们使用 Kubernetes 统一调度资源，所以启动前并不知道各个任务会被调度到哪台机器运行。好在 gRPC 服务以 URL 作为入口，天然支持 IP 和域名两种地址形式。因此，我们只要在作业启动前预先设置好所有任务的 gRPC 服务域名，并将域名绑定到任务角色而非具体的主机，任务在运行时就能够顺利找到彼此。为了对这种动态绑定到任务角色的域名进行解析，我们需要使用 Kubernetes 的 kube-dns 服务。这样一来，我们不再需要在启动任务时手动指定各个任务的主机名或 IP 地址。
- 为了能够及时回收 TensorFlow 分布式训练作业占用的资源，特别是不能自动结束的PS任务所占用的资源，我们可以使用 Kubernetes Dashboard（https://github.com/kubernetes/dashboard）监控所有 Pod 的运行状态。当我们发现所有 worker 任务都已经运行结束时，就可以调用`kubectl`工具删除包括 PS 任务在内的整个分布式训练作业。用户亦可通过编写脚本，调用 Kubernetes 的命令行工具或 API，将这一过程自动化。

综合以上思路，图 15-3 给出了在 Kubernetes 集群中部署 TensorFlow 分布式训练作业的一个实例。需要注意的是，ecosystem 项目给出的 Jinja 模板将 worker 任务设置为 ReplicaSet，而我们建议将其设置为 Pod。这是因为 ReplicaSet 会始终保持运行状态，即使 worker 任务成功结束，也会被立即拉起再次运行。这使得我们无法轻易识别出已完成的 worker 任务。

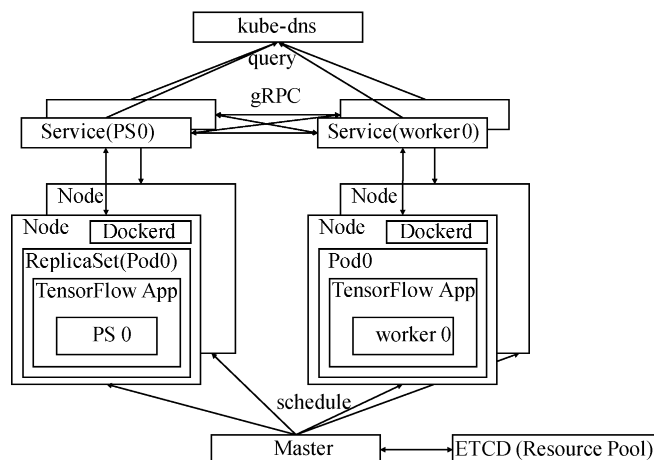


图 15-3 TensorFlow 分布式训练作业在 Kubernetes 集群中的逻辑部署图

TensorFlow与Kubernetes的结合是深度学习计算库与分布式容器管理系统两个生态圈的有机结合。这一结合既弥补了计算库本身在运行时和服务管理方面的不足，又促进了分布式容器管理系统在深度学习应用领域的发展。目前，国内外许多公司的人工智能公有云和私有云服务都选择将 Kubernetes 作为资源调度引擎，将 TensorFlow 作为深度学习计算库。相信随着这一波人工智能浪潮的兴起，Kubernetes 社区与 TensorFlow 社区的互动将会更加频繁。

15.4　TensorFlow 与 Spark 生态的结合

Apache Spark 作为一套高性能的分布式大规模数据处理框架，已在大数据领域得到了广泛的应用。Spark 的架构设计注重易用性与通用性，因此其生态系统也随着计算模式的发展变化而不断壮大。除了经典的有向无环图（DAG）迭代计算框架，Spark 平台还支持接入交互式处理（Spark SQL）、流计算（Streaming）、机器学习（MLlib）、图计算（GraphX）、统计分析（SparkR）等多种模式的计算引擎。TensorFlow 等面向深度学习的计算引擎同 Spark 结合自然成为一个有价值的工作方向。

相比大数据生态中的其他平台软件，TensorFlow 尚存在两点不足：其一，从 Google 内部分布式计算框架中剥离出来的 TensorFlow 开源版本核心组件只是一套开发库和计算引擎，尚缺乏系统而完善的运行时管理机制和透明的分布式资源管理能力。例如，它没有提供作业和任务的排队调度框架，其分布式应用也需要在运行时显式指定集群 IP 地址和 GPU 资源。其二，TensorFlow 支持的数据源与输入、输出数据结构并不完全兼容以 Apache 社区软件为代表的大数据生态。尽管 TensorFlow 已经支持 HDFS 等分布式文件系统访问，然而它仍不能支持大数据平台中的一些内存数据结构，这对于多种计算引擎协同处理同一数据集的应用场景并不友好。TensorFlow 若能与 Spark 对接，恰恰可以通过 Spark 的资源调度框架和 RDD（Resilient Distributed Dataset）数据结构弥补这两点不足。

Yahoo!公司的 Big ML 团队于 2017 年年初开源了 TensorFlowOnSpark 项目（https://github.com/yahoo/TensorFlowOnSpark）。该项目旨在将 TensorFlow 计算作业运行在 Spark 集群上，在调度和数据两方面打通 TensorFlow 与 Spark，从而使得 Spark 平台能够无缝支持深度学习，同时提升 TensorFlow 在分布式环境中的易用性。图 15-4 展示了 TensorFlowOnSpark 及相关组件在 Spark 集群中的部署结构。

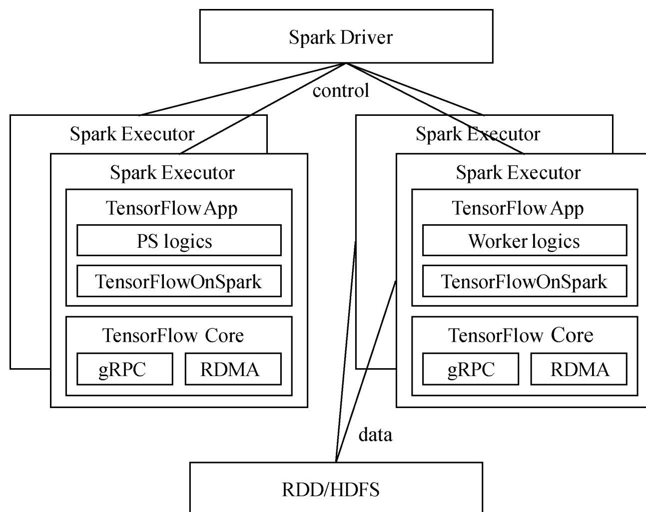


图 15-4　TensorFlowOnSpark 及相关组件在 Spark 集群中的部署结构

TensorFlowOnSpark 以 Python 语言实现，本质上是一套符合 Spark 应用框架规范的 Python 开发库。基于该库开发的 Python 应用程序可以直接由 Spark 集群的 Executor 加载运行，接受 Spark Driver 的调度并共享 RDD 中的数据。在调度方面，TensorFlowOnSpark 提供对接 Spark 能力的 `TFSparkNode` 等组件，用于实现端口和 GPU 分配、任务启停等功能；提供面向开发者的 `TFCluster` 等编程抽象，具有简单明了且与资源解耦的接口，用于集群配置、模型训练和推理等操作。在数据方面，TensorFlowOnSpark 提供两种提取训练和推理数据的方式：使用 TensorFlow 原生的文件读入接口与 `QueueRunner` 机制，从 HDFS 文件获取数据；使用 TensorFlow 的 `feed_dict` 机制，从 Spark Executor 加载的 RDD 中获得数据。

现有的 TensorFlow Python 应用程序需要少量修改，以便集成 TensorFlowOnSpark。修改后的程序可以通过 `spark-submit` 命令启动，运行在多种计算引擎共享的 Spark 集群中。TensorFlow OnSpark 的开发者给出了 MNIST、CIFAR-10、Inception 等模型训练、推理程序的修改示例，体现了该框架的易用性与通用性。

除了 TensorFlowOnSpark，亦有其他一些开源项目关注 TensorFlow 与 Spark 的结合。

- Arimo 公司开源的 TensorSpark 项目（https://github.com/adatao/tensorspark）提供了一套可运行于 Spark+Yarn 集群环境的分布式模型训练框架。它实现了一种不同于 TensorFlow 原生分布式模式的参数服务器架构，并提供了一组典型模型的移植示例。该项目基于 Python 开发，逻辑相对简单，适合于二次开发者学习如何桥接 TensorFlow 与 Spark 生态系统。
- TensorFlow ecosystem 子项目之一的 spark-tensorflow-connector（https://github.com/tensorflow/ecosystem/tree/master/spark/spark-tensorflow-connector）聚焦于 TensorFlow TFRecords 与 Spark DataFrame 两种数据格式的相互转换。DataFrame 是 Spark SQL 提供的结构化数据编程抽象，以命名列方式组织分布式数据集。将其作为 TensorFlow 的数据源，有利于数据的流水线式处理和数据集的复用。
- Databricks 公司开源的 TensorFrame 项目（https://github.com/databricks/tensorframes）则为 TensorFlow 提供直接操作 DataFrame 数据格式的能力。它具有 Tensor——DataFrame 映射机制，支持归约与聚合算法，并提供一套简单的 TensorFlow Scala 语言 API，旨在从数据和编程层面连接两个生态。TensorFrame 的内存计算机制也是一种遵循 Spark 理念、性能优势明显的设计。

可以预见，随着应用模式的扩展和数据规模的扩张，TensorFlow 等深度学习计算引擎必然会与 Spark 等大规模数据处理平台发生越来越频繁的交互。数据的量变终将带来智能的质变，两个生态的结合点迸发的火花值得相关研究者与开发者密切关注。

15.5 TensorFlow 通信优化技术

在并行和分布式系统中，通信性能是决定软件整体性能的重要因素。TensorFlow 社区在不断改善平台计算能力的同时，也不忘对平台通信能力进行优化，使之高效利用硬件带宽，降低消息

延迟，达到提升应用执行速率和并行加速比、增强软件可伸缩性的目标。TensorFlow 生态系统中的通信优化技术既包括利用高性能硬件特性加速数据传输的工作，也包括改进分布式模型与通信算法的实践。我们在 13.4 节中分析了已被 TensorFlow 社区采纳的 Yahoo! RDMA 通信模块，这里另外介绍 TensorFlow 生态中的几项通信优化技术。

随着数据和模型规模的日益增长，深度学习应用对硬件能力的需求不断提高。为了在提升计算能力的同时尽可能少地引入网络通信开销，很多硬件厂商力推单服务器多 GPU（即“单机多卡”）的解决方案。对于这种环境，实现多 GPU 内存之间的数据高效传输是确保计算得以线性加速的重要条件。在高性能计算领域广为应用的集合通信算法，如全局归约（allreduce）、全局收集（allgather）、广播（broadcast），很适合解决单机多卡环境下的神经网络参数归约计算与跨设备交换问题。然而，传统的集合通信算法没有考虑单机多卡数据链路的拓扑特点，一些逻辑上复杂度较低的算法在物理实现时却有可能造成 PCIe 总线竞争，性能反而不佳。为了解决这个问题，NVIDIA 公司推出了基于 CUDA、适用于单服务器多 GPU 内存的集合通信库 NCCL（https://github.com/NVIDIA/nccl），它提供了一系列与单机多卡数据链路相适配的集合通信函数。图 15-5 给出了 NCCL 中的 broadcast 算法执行过程示例。可以看出，在实线所示的物理链路上，算法以虚线所示的环序串行方式传播数据。尽管其复杂度高于树状并行算法，然而能够避免总线出现拥塞。为了让 TensorFlow 利用 NCCL 技术加速单机多卡通信，NVIDIA 贡献了一套集合通信操作及其对应的算法核函数。它们已被放置在 TensorFlow 源代码包的 tensorflow/contrib/nccl 目录下。受到 TensorFlow 现有的优化器设计限制，NCCL 尚不能实现对已有应用程序的透明加速。用户需要修改数据流图逻辑，以便利用这一高效的通信机制，充分发挥 GPU 内存带宽的潜力。

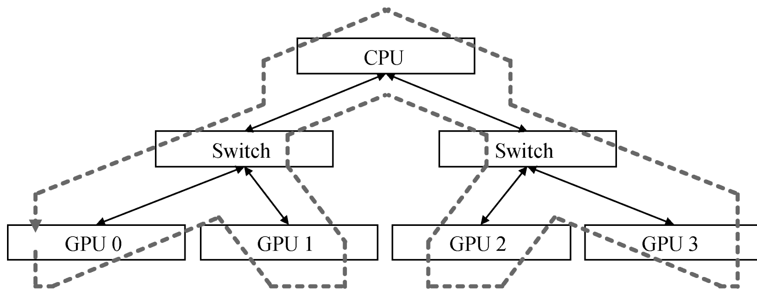


图 15-5　NCCL 中的 broadcast 算法执行过程示例

在多机分布式计算场景下，集合通信同样是优化神经网络算法的选项之一。TensorFlow 生态系统中亦有一些分布式集合通信方面的改进工作，这里介绍一个 allreduce 算法的应用实例。基于环状消息传递的 ring allreduce 算法在高性能计算领域具有悠久的历史，已被 MPI 等通信库广泛采纳。如图 15-6 所示，该算法通过分发–归约（scatter-reduce）和全局收集（allgather）两个步骤，实现归约计算任务的分布式执行。百度公司研究团队开源的 TensorFlow-Allreduce 项目（https://github.com/baidu-research/tensorflow-allreduce）将这一算法引入了 TensorFlow。不同于 Yahoo! TensorFlow-RDMA 那种接入高端硬件的透明加速方案，这个方案在 TensorFlow 中引入了

新的操作节点（`MPIAllreduceOp`、`MPIAllgatherOp`）和优化器（`DistributedOptimizer`）类型，并基于 MPI 库的进程管理能力设计了一套分布式运行时机制。用户基于这些接口编写的深度学习模型训练代码能够通过 ring allreduce 算法实现高效的参数通信和归约计算。此方案应用于带有 InfiniBand 网络和 GPU 的集群环境时，MPI 库内置的 RDMA 通信及 GPUDirect RDMA 技术能够进一步发挥硬件的潜能。不过，集合通信的引入使得参数更新过程具有类似于 BSP（Bulk Synchronous Parallel）模型的全局同步特征，这在一定程度上影响了程序的灵活性和容错性。在实际应用中，开发者可以依据算法的需求，合理选择分布式通信框架。

将 MPI 库作为 TensorFlow 通信框架使用的第三方项目，除了百度的 TensorFlow-Allreduce，还有来自美国的西北太平洋国家实验室（PNNL）的 MaTEx-TensorFlow。该项目是 MaTEx 高性能机器学习软件包（https://github.com/matex-org/matex）的一部分，这个软件包旨在利用高性能计算集群（如 TOP 500 超级计算机）加速各类机器学习、深度学习应用。MaTEx-TensorFlow 在 TensorFlow 的 Python API 层插入了 MPI 通信原语，包括在 `Variable` 抽象中实现数据分发的 broadcast 操作以及在梯度函数中实现归约计算的 reduce 操作等。它也允许用户直接在 Python 应用层使用封装过的集合通信操作，以便精确控制归约计算逻辑。对于符合 MaTEx-TensorFlow 约束条件的单机风格应用程序，用户只需要插入必要的数据集读取代码，就可以经由 MPI 进程管理器来启动程序，实现数据并行的多进程分布式计算。对于在高性能集群环境中加速既有单机 TensorFlow 应用的需求，MaTEx-TensorFlow 不失为一种简单易用的选择。官方测试结果显示，它能够为既有应用带来近线性的并行加速比。不过，MaTEx-TensorFlow 在编程方式上有一定的局限性，它同样不是一种对既有应用完全透明的通信优化方案。

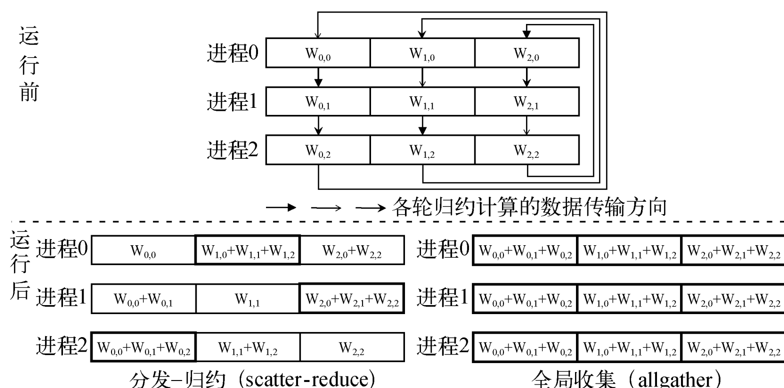


图 15-6　ring allreduce 算法的执行过程示例

可以看出，来自高性能计算领域的高端通信技术与 TensorFlow 平台有多种结合的可能性，不同的方案各有所长。当前，很多互联网和云计算公司已在生产环境中引入了多 GPU 服务器和 InfiniBand 网络等基础设施。要想发挥这类高端硬件的潜力，软件层能力的跟进是必要条件。在这样的技术背景下，TensorFlow 开源版本在通信优化方面尚有广阔的发展空间。

15.6　TPU 及神经网络处理器

在计算机科学的发展史上，每当有新的应用需求爆发时，如果通用硬件的能力不能在短时间内以合理的成本满足应用需求，往往就会有专用硬件问世。这个规律在机器学习、深度学习领域也不例外。近年来，人工智能技术在互联网、大数据的推动下，迎来了新的发展浪潮。面对以神经网络训练和推理算法为代表的计算需求，现有的 CPU、GPGPU 等通用计算器件在性能、性价比、能耗效率等方面的劣势逐渐显现。不少商业巨头和学术机构将目光转向了专用硬件研发。在这样的背景下，专用硬件也成为 TensorFlow 生态系统中举足轻重的成员。

Google 公司自 2015 年起，在其内部的 TensorFlow 集群上使用名为 Tensor Processing Unit（TPU）的定制化硬件运行神经网络推理（预测）任务。在 2017 年的计算机系统领域顶级学术会议 International Symposium on Computer Architecture（ISCA）上，Google 的研究人员发表了题为“In-Datacenter Performance Analysis of a Tensor Processing Unit”的论文，其中介绍了 TPU 的部分设计和实现细节。在 2017 年的 Google I/O 大会上，面向云服务场景、同时支持神经网络训练和推理任务的 TPU 2.0 版本（又称 Cloud TPU）公之于众。

TPU 芯片是一种基于复杂指令集（CISC）、运行时可编程的微处理器，属于专用集成电路（ASIC）范畴。TPU 芯片以 PCIe 板卡形式连接到服务器，板卡上亦集成了内存（DRAM）芯片。对于软件开发人员而言，TPU 的编程使用模式与 CPU、GPU 相似，具备一定的领域内“准通用”特征，不同于需要针对特定算法而烧写程序的 FPGA。第一代 TPU 的芯片布局如图 15-7 所示（图中方框的位置与面积大体反映实际芯片的布局），该芯片包含矩阵乘法运算器、统一缓冲区、激活流水线、控制器、累加器，以及 DRAM 接口和 PCIe 接口等单元。由芯片设计即可看出，TPU 以矩阵作为基本数据元素，以矩阵乘法作为关键运算指令。这一方面契合了神经网络推理算法的核心计算需求，另一方面也是传统高性能并行机、向量机设计思想的升华。这一代 TPU 并不通过多核来提升并行性，它的并行性设计类似于向量计算机：矩阵乘法运算器通过一组先入先出（FIFO）队列从 DRAM 中并行读取多个数据，在同一计算周期内完成 64K 次累加。这种设计使得其控制单元远比多核 CPU 或 GPU 简单，仅占芯片 2% 的面积。Google 宣称，使用 TPU 替代传统硬件，在神经网络推理算法上加速约 15 ~ 30 倍，能耗效率提升约 30 ~ 80 倍。

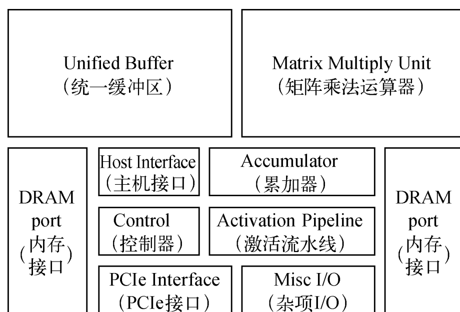


图 15-7　第一代 TPU 的芯片布局

TPU 2.0 则在浮点运算方面做了大幅增强——这正反映了神经网络训练过程的核心需求。这一代 TPU 以独立的“TPU device”板卡形式部署，每块板卡上集成 4 个计算芯片，提供高达 180 TFLOPS 的浮点运算能力。多块板卡协同工作时，可以构成 TPU pod 部署模式，比单板能够实现的运算能力有近百倍提升。Google 公司内部的机器翻译应用实例表明，TPU 2.0 的模型训练性能显著高于分布式 GPU 集群。目前，两代 TPU 均不对外销售，而是作为 Google 云引擎的后台计算能力实现者，通过支撑 PaaS 和 SaaS 服务为 Google 公司创造价值。

面对人工智能与神经网络处理器的巨大潜在商机，Google 之外的不少业界大鳄也都蜂拥而至，与之相关的创业公司亦如雨后春笋般兴起。对 TensorFlow 等主流机器学习、深度学习平台的无缝支持，往往会成为这类专用硬件的重要卖点。2016 年，Intel 公司收购了 Nervana——一家在深度学习算法与芯片方面掌握核心竞争力的公司。随后，Intel 宣布了其人工智能芯片路线图，其中包括面向深度神经网络计算的 Lake Crest 家族，以及与 Xeon 处理器技术紧密结合的 Knights Crest 家族。这些处理器将引入“基于张量的架构”，能够为 TensorFlow 等平台提供加速支持。在神经网络训练态加速领域独占鳌头的 NVIDIA 公司也并未放松对 TPU 的警惕，该公司已于 2017 年宣布了对标 TPU 的深度学习加速器（Deep Learning Accelerator，DLA），期望构建高性能、低功耗的神经网络推理态解决方案。NVIDIA 计划将 DLA 的设计开源，可以预计 DLA 将与 TensorFlow 的开源生态产生良性互动。

中国在神经网络处理器研发方面处于相对领先的地位。中国科学院计算技术研究所在 2014 年至 2016 年发表的 Diannao 系列人工智能体系结构论文，在计算机系统结构学术圈引起了强烈反响。这些成果不但在深度学习指令集等研究方向上开创了世界第一，而且在图像处理等应用领域取得了良好的使用效果。起源于该团队的寒武纪公司则致力于推动人工智能芯片的产品化和商用化，通过 IP 授权、芯片服务、智能子卡等解决方案支撑人工智能技术融入应用场景。寒武纪芯片的指令集能够支持包括 TensorFlow 在内的计算引擎，这在降低专用硬件使用门槛的同时，也拓展了 TensorFlow 自身的硬件生态。此外，中星微公司发布的“星光智能一号”嵌入式神经网络处理器也实现了对 TensorFlow 神经网络模型的支持，能够应用于视频监控等领域。可以预见，神经网络处理器的大戏才刚刚揭开帷幕，未来该领域还会有更多激动人心的成果涌现。TensorFlow 在这个生态系统中既是一个推动者，也会是一个受益者。

15.7 NNVM 模块化深度学习组件

近几年，来自学术界和互联网公司的机器学习、深度学习框架层出不穷。不同框架通常具有各自的优势领域和适用范围，因此也聚焦了各自不同的用户群体。在日趋复杂的应用场景中，如何能够扬长避短地使用多种计算框架、降低系统部署和学习成本，成为平台层软件设计者关心的问题。同时，新兴应用模式和专用硬件的涌现对计算框架不断提出新的需求，使得计算框架开发的复杂度持续增长。在这种情况下，如何提升计算框架的开发效率、增强计算框架对上下层演进的应变能力，也是一个重要的工程课题。TensorFlow 的 XLA 正在探索这个方向。与此同时，开

源社区和第三方厂商也不甘示弱。

NNVM（https://github.com/dmlc/nnvm）是 DMLC 开源社区于 2016 年年底提出的一套模块化深度学习组件，它期望通过提供一种类似于编译器领域 LLVM 框架的中间表示层（Intermediate Representation），将深度学习平台的前端编程接口与后端硬件加速机制解耦。一方面，允许多种编程框架的接口工作在同一套运行时核心之上，降低系统软件的部署开销和开发人员的学习成本；另一方面，前后端开发者聚焦于各自关注的领域，这有利于深度学习平台的定制化开发及相关生态系统的快速发展。为了展示 NNVM 的能力，DMLC 发布了一套模拟 TensorFlow 框架的示例性项目——TinyFlow。它不但展现了 NNVM 力图为深度学习系统开发提供新范式的雄心，而且为 TensorFlow 生态系统注入了新的设计思路。

图 15-8 给出了 TinyFlow 及 NNVM 的主要组件逻辑结构。作为 DMLC 工具集的一部分，TinyFlow 和 NNVM 复用了 DMLC Core 的部分组件。DMLC Core 是 DMLC 社区的公共基础开发库，已在 MXNet 等项目中应用。它包含一组专为深度学习需求设计的 C++ 数据结构与实用工具类，涉及分布式文件系统访问、参数表示与处理、算子注册与管理等功能。在中间层组件分工方面，NNVM 并没有包揽前后端的所有抽象。NNVM 本身主要关注计算图的中间表示，它使用同样来自 DMLC 社区的 TVM 组件实现张量算子在不同硬件平台上的编译优化。TVM 借助一种名为 HalideIR 的中间表示层表达代数算法的语义，并能够将这些计算语义高效转换为 x86、ARM、CUDA、OpenCL 等多种后端设备的原语。

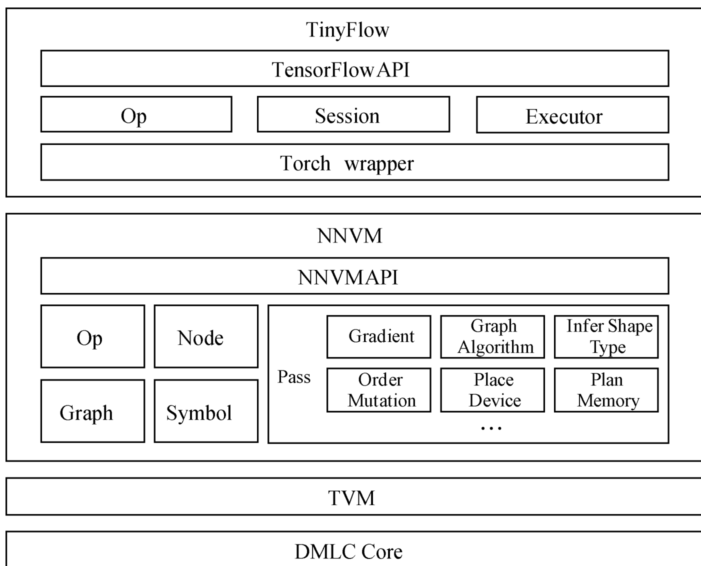


图 15-8　TinyFlow 及 NNVM 的主要组件逻辑结构

NNVM 以计算图作为中间表示层的核心抽象，期望提供统一的、独立于前端编程接口和后端硬件实现的计算图表示、优化和执行机制。它继承了现有深度学习平台以增加算子方式获得应

用层开发灵活性的惯例，同时借鉴了编译器领域“优化遍”（optimization pass）的思想。通过引入一组模块化的 pass 函数，增加系统层开发的灵活性并提升运行时性能。基于这个思路，NNVM 提供一套基于 C++ 实现的、具有共性特征的数据结构，以及抽象为 pass 函数的工具类。数据结构主要包括 `Graph`、`Op`、`Node`、`Symbol` 等计算图相关的通用类型。pass 函数主要涉及图上的梯度计算、形状推导、控制依赖管理，以及图的内存分配和算子的设备分配等功能。这些 pass 函数以独立功能模块的形式存在，尚不构成闭环的计算图优化和执行引擎。

TinyFlow 作为示例性项目，以 NNVM 和 Torch 为基础，实现了一套与 TensorFlow API 子集兼容的简化版深度学习系统。TinyFlow 内部使用 NNVM 定义的计算图数据结构，基于 Python 封装面向应用开发的 API，提供 TensorFlow 的 `Variable`、`Session` 等基础抽象以及 `GradientDescentOptimizer` 等基础优化器。由于 NNVM 不关注算子的逻辑，其图执行引擎也尚未成型，TinyFlow 为了简化开发，封装了 Torch 的一部分算子，并开发了面向 Torch 算子的执行器与会话管理组件。这些组件遵循 NNVM 的计算图编程范式，利用 NNVM 的内存分配、形状推导等 pass 函数。TinyFlow 的初始发布版本只提供单机执行模式，尚不支持分布式执行。其总代码量约 2000 行，远小于 TensorFlow。其中半数以上的代码是对 Torch 算子的封装，以及执行器与会话机制对 Torch 的适配。

NNVM 这类中间层技术的出现，从计算机系统软件的发展历史看具有必然性。它为深度学习系统的开发提供了模块化、去中心化的新思路。尽管 NNVM 当前的设计和实现仍处于起步阶段，然而我们不能小视这种开发范式对深度学习生态系统的影响。在 DMLC 社区积极推动、MXNet 等平台逐步向 NNVM 迁移的背景下，NNVM 有望成长壮大。TinyFlow 作为一个示例，不仅是在验证 NNVM 的能力，也是对 TensorFlow 生态的拓展。TensorFlow 内部组件耦合度较高、发展方向受制于商业公司等为人诟病的问题，或许会随着中间层、模块化技术发展和第三方兼容实现的出现而得到解决。

15.8 TensorFlow 未来展望——TFX

TensorFlow 作为当下最热门的机器学习开发库，其未来发展一直牵动着全球开发者的心。与 Hadoop 和 Spark 等成熟的大数据系统不同，TensorFlow 仅仅是一个开发库。它为用户提供了灵活的接口，使得用户可以按照自身需求实现各种不同类型的机器学习模型。然而，TensorFlow 并不提供具有完整运行时框架的端到端解决方案。用户如果想要基于 TensorFlow 搭建一套易用且可靠的机器学习系统，还需要不少的开发和适配工作。类比已有的成熟系统，一套完整的机器学习系统主要由以下几个模块构成：数据分析、转换和验证、模型训练、微调和评估，以及模型服务发布和升级。

通常，大多数用户是为某些特定场景和需求而开发一套机器学习系统，其中难免出现许多胶水代码和适配脚本，这将使得系统本身变得更加脆弱。比如，新收集的不符合规范的数据导致持续训练过程出错，新升级的模型准确率下降导致服务质量降低等。如果用户没有考虑到类似问题

并作出对应的设计，那么将会背上更多的技术债（technical debt），这使得普通用户难以实现和维护一套易用且可靠的机器学习系统。为了解决以上的种种问题，同时以一种尽可能通用的方式来设计和实现基于 TensorFlow 的机器学习系统，Google 公司提出了它们的解决方案——TensorFlow Extended（TFX）。

Google 在 2017 年的 SIGKDD 会议论文（TFX: A TensorFlow-Based Production-Scale Machine Learning Platform）中正式介绍了他们在 TFX 上的一些工作进展。图 15-9 给出了 TFX 的高层组件逻辑架构。TFX 是集成数据、模型和服务模块的通用机器学习平台。平台中各模块共享包含作业管理、监控和调测，以及数据、模型和评估指标可视化的统一前端，使用统一的环境配置和作业编排能力，共享垃圾回收、数据访问控制和公共的基础设施服务。相比用户根据特定需求开发的机器学习系统，TFX 一方面简化了平台环境配置，减少了用户自定义代码的开发，加快了模型训练和验证流程；另一方面有效结合了 TensorFlow 生态与大数据系统生态各自优势，缩短了模型服务发布时间，提升了平台的通用性和易用性。

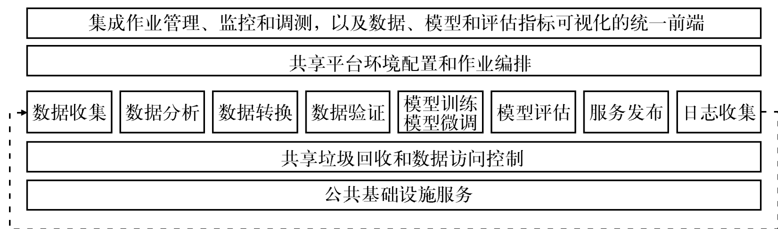


图 15-9　TFX 的高层组件逻辑架构

为了展示 TFX 带来的诸多好处和实际价值，它的作者在 Google Play 应用商店部署了一套 TFX。每当用户下载或评论某个应用时，新产生的数据就会被 TFX 的日志模块收集。这些数据经过处理后进入模型持续训练流程，成为新的训练样本。经过一段时间的增量学习后，模型的准确率也会得到提升。同时，使用新模型升级的线上服务的质量也会不断提高，促使用户下载更多符合自身喜好的应用。得益于 TFX 端到端的数据、模型和服务全流程处理能力，Google Play 商店的应用装机数上升了 2 个百分点。

回望过去，我们不难发现 Google 正在通过投入大量时间和精力，在 AI 领域广泛布局。在算法模型和应用服务方面，TensorFlow 已基本确立领导地位和领先优势；在基础设施和运行时管理方面，Kubernetes 已成为容器编排领域的事实标准；在 AI 芯片方面，TPU 已落地并同时支持端侧推理和云侧训练任务；在人才储备和科学研究方面，Google AI 中国中心也已正式建立。

展望未来，依托于 TPU 提供的高效计算能力，以及 Kubernetes 和 Borg 提供的自动化调度能力，Google 内部闭源的 TFX 有望通过云平台向全球开发者提供一站式机器学习服务。经过 2015 年至 2017 年的积累，机器学习框架已经非常丰富。2018 年将会是机器学习平台爆发的元年。我们有理由相信在 Google 的刺激和引领下，全球开发者将能够体验到更加便捷的机器学习开发环境和模型服务运作模式，而广大用户也将能享受到各行业不断推出的智能服务所带来的舒适体验。

15.9 小结

生态环境的开放性是开源软件的重要魅力，也是其获得商业公司和技术高手青睐的重要特质。TensorFlow 就是这样一款高度重视生态建设的开源软件，它的社区托管了大量来自 Google 公司和第三方开发者贡献的生态系统组件，涉及算法模型、实用工具、运行时环境支持等。与此同时，TensorFlow 团队对第三方独立维护的衍生版本或周边项目持欢迎态度，大量面向不同市场需求和应用场景的 TensorFlow 相关项目应运而生。在软件平台方面，Keras 算法模型库、Kubernetes 容器调度器和 TensorFlowOnSpark 生态对接组件是具有代表性的生态系统成员；在专用硬件方面，Google 的 TPU 及国内外诸多厂商各自研发的神经网络处理器为 TensorFlow 生态系统的扩张增添了筹码。随着互联网、大数据和人工智能技术向各类应用领域乃至人类日常生活的深入渗透，我们可以预见，作为核心计算引擎之一的 TensorFlow 具有广阔的发展前景，有望成为一代经典的开源软件。

附录 A 常见问题解决方案

Q：在 Python 解释器中执行 `import tensorflow` 命令时出现 `ImportError: No module named platform` 或类似的错误，即无法找到 tensorflow 包内部的模块。

A：这类错误通常出现在编译安装并运行 TensorFlow 的场景。如果 Python 解释器的当前工作目录是 TensorFlow 源代码目录，其中恰好存在同 TensorFlow 安装目录类似的结构，Python 解释器的模块查找就会出现问题。这种情况下，用户只需要使用 `cd` 命令将当前工作目录切换到其他位置，重新运行 Python 解释器即可。

Q：如何在使用 HTTP 代理服务器的环境中编译 TensorFlow？我已将代理服务器的 CA 证书导入操作系统和 Java 环境，为什么 Bazel 仍然不能通过代理服务器下载文件？

A：在企业局域网等使用 HTTP 代理服务器的环境中编译 TensorFlow 时，Bazel 下载部分第三方软件的过程可能涉及访问 HTTPS 协议的网站。代理服务器不能提供有效的 SSL 证书，因而会被 Bazel 所调用的 Java 网络客户端认定为非法，从而阻止下载。针对这种情况，用户需要使用支持 SSL 证书动态伪造特性的 HTTP 代理服务器。这类 HTTP 代理服务器均会为用户提供一个 CA 证书，用户需要将其导入到客户端的信任证书列表中。有经验的读者可能知道如何将 CA 证书导入 Windows、Linux 操作系统或 Java 环境，但 Bazel 的特殊之处在于：它默认使用的 Java 环境是一套自带的 JDK，其信任证书来自这套 JDK。Bazel 的完整性检查机制会阻止用户对其中的证书进行修改。为了解决这个问题，用户可以从 Bazel 官方网站下载 without-jdk 的版本。这种版本的 Bazel 使用的是当前操作系统环境变量所指定的 Java 环境。此时，用户只需要调用 `keytool` 命令，将代理服务器的 CA 证书导入外部 Java 环境，即可使得 Bazel 正常访问 HTTPS 链接，从而顺利编译 TensorFlow。

Q：在阅读 TensorFlow 源代码时，发现有的地方引用的某些类型、方法或常量在语义上明显属于 TensorFlow，在源代码中却无法找到。它们到底是在哪里定义的？

A：TensorFlow 的源代码中，多处使用了编译时的代码动态生成机制。例如通过 Protocol Buffers 和 gRPC 编译器，将 .proto 文件编译为 C++或其他语言的数据结构和通信接口层代码；又如使用 SWIG 工具，将 .i 文件编译为 Python 语言访问 C 或 C++库的接口层代码。除了使用这些外部工具，TensorFlow 内部也设计了一些编译时代码生成机制，典型的例子是生成以 `gen_` 开头的算子模块时所使用的自定义 Bazel 构建规则——`tf_gen_op_wrapper_cc`。读者在遇到无法从源代码中找到某些定义的问题时，不妨考虑从这些角度探索。



微信连接



回复“TensorFlow”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群Ⅰ：218139230
图灵读者官方群Ⅱ：164939616

图灵社区

iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

一本科技读物通常难以在理论、源码和实践之间取得平衡：理论过多很难让读者学以致用并付诸实践，源码过多容易让读者望而却步且只见冰山一角，而过于聚焦实践案例则难以帮助读者举一反三，参透更高层的设计理念与哲学。本书却少有地由浅入深，既完整介绍了深度学习和TensorFlow的技术演化、生态全貌、设计理念，又及时地在一段理论陈述和数学原理之后通过源码层面的案例分享帮助读者将理论落地。为了帮助读者举一反三，知其然还知其所以然，本书不光传授了TensorFlow技术本身，还系统地介绍了理解和掌握TensorFlow所需的周边知识，用“自包含”的方式为读者提供了“一站式”的从入门到精通的指引。

—— 张鑫，才云科技创始人兼CEO

TensorFlow是一个深度学习的基础框架，自2015年年底开源以来，它被不断应用到各个领域当中，也逐渐孕育出了一个活跃的开源社区。本书的作者就是这个社区的贡献者，他们对TensorFlow有深层次的理解。本书从独特的角度剖析了TensorFlow和分布式TensorFlow的运行机理，并以TensorFlow 1.2为基础，用简单易懂的语言讲解了TensorFlow的安装、模型编写、可视化一直到生产环境部署的方方面面，同时穿插介绍了深度学习的基本概念。该书是作者的呕心沥血之作，是一本非常值得阅读的TensorFlow中文书。

—— 周玥枫，Google Brain资深工程师

在深度学习技术进入商业化实用阶段，这本书通过技术概念和实践案例讲解，对广大AI技术爱好者深度了解和应用TensorFlow技术的本质内涵、技术框架和应用体系提供重要参考，值得一读。

—— 夏命榛，华为人工智能领域主任工程师

本书以TensorFlow为线索介绍深度学习的算法和系统，既包含算法的背景知识，又囊括系统的实现原理，并给出TensorFlow中的代码示例，是综合算法理论和系统原理并支持动手实践的佳作。

—— 邹永强，云账户联合创始人兼CTO

当前人工智能的发展高度依赖数据、算法和计算能力三要素。在计算能力越来越强的今天，数据和算法成为人工智能发展的两个关键要素。而作为人工智能的核心算法，深度学习对于技术工程师来说依然迷雾重重。TensorFlow是深度学习领域非常重要的开源框架，基于TensorFlow的应用越来越广泛地应用到安防、电商、金融、医疗等领域，也正在逐步渗透到工业领域。本书从底层技术入手，深入浅出地讲解了TensorFlow的原理、架构、核心算法和应用场景，并且展示了其强大的生态配套体系，是不可多得的TensorFlow学习教材。未来已来，让我们积极拥抱人工智能的未来。

—— 周公爽，博拉科技创始人兼CEO

这是一本来自工业界技术专家的书，作者对深度学习框架和机器学习算法有多年的深入研究，对TensorFlow在业界的实战应用有丰富的经验和独到的见解。本书系统详尽地介绍了TensorFlow的主要模块及使用方法，同时介绍了CNN、GAN和RNN等深度学习算法模型和TensorFlow的内部核心模块。本书一气呵成，深入浅出，每章均配有总览流程图和详细的案例代码，特别适合工程师和研究者入门。

—— 王锦鹏，微软亚洲研究院助理研究员

本书由浅入深，详细介绍了TensorFlow的编程方法与工作原理。本书在介绍深度学习与TensorFlow基本概念和用法的同时，深入分析了TensorFlow的系统架构与实现原理，是TensorFlow开源系统贡献者的重要参考资料。

—— 郑泽宇，才云科技顾问

图灵社区：iTuring.cn
微　　博：@图灵教育　@图灵社区

分类建议　计算机 / 人工智能 / 深度学习/TensorFlow

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-48094-1
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks